Matthias Hert

OntoAccess

RDF-based Read and Write Access to Relational Databases

Dissertation

Advisors

Prof. Dr. Harald C. Gall
University of Zurich

Prof. Abraham Bernstein, PhD
University of Zurich

University of Zurich[UZH]

OntoAccess

RDF-based Read and Write Access to Relational Databases

A dissertation submitted to the Faculty of Economics, Business Administration and Information Technology of the University of Zurich

for the degree of
Doctor of Science

by
Matthias Hert
from
Messen, SO, Switzerland

Accepted on the recommendation of
Prof. Dr. Harald C. Gall
Prof. Abraham Bernstein, PhD

2012

The Faculty of Economics, Business Administration and Information Technology of the University of Zurich herewith permits the publication of the aforementioned dissertation without expressing any opinion on the views contained therein.

Zurich, March 2012

Head of the Ph.D. committee for informatics:
Prof. Abraham Bernstein, PhD.

Acknowledgements

It is my pleasure to thank all people who accompanied me during my PhD.

First of all, I would like to thank Harald Gall for giving me the opportunity to pursue a PhD in the s.e.a.l. group, even though my interests were not closely related to the main research focus of the group. I am also grateful for his support in the difficult times of my PhD studies. I thank Avi Bernstein for accepting to be my co-advisor.

Special thanks go to Geri Reif for introducing me to the Semantic Web in the first place, for the many discussions we had about my work, but most importantly for his continued advice even after leaving academia. Geri, I am very grateful to know you and to have worked with you.

Many thanks to my fellow PhD students, particularly to Michael Würsch for repeatedly "breaking" the Semantic Web resulting in interesting debugging sessions and discussions. To Amancio Bouza, Emanuel Giger, Giacomo Ghezzi, Martin Brandtner, Sandro Buccuzzo, Beat Fluri, and all other s.e.a.l. members. I also want to thank my friends in the DDIS group, particularly the members of the "UZH delegations" at Reasoning Web 2008 and ESWC 2009, Jonas Tappolet and Tom Scharrenbach.

I thank my family for unconditionally supporting me not only during my PhD studies but always. Finally, I would like to express my deepest gratitude to my girlfriend, for being with me and for supporting me in so many ways.

Matthias Hert
Zurich, March 2012

Abstract

Relational Databases (RDBs) are used in most current enterprise environments to store and manage data. While RDBs are well suited to handle large amounts of data, they were not designed to preserve the data semantics. The meaning of the data is implicit at the application level but not explicitly encoded in the relational model. The Semantic Web provides a common framework that allows data to be shared and reused across application, enterprise, and community boundaries. Although developed for the Web, these Semantic Web technologies have proven to be useful in other domains as well, especially if data from different sources has to be exchanged or integrated. In existing systems, however, it is not always possible or desirable to convert all relational data to RDF as other business-critical applications rely on the *relational* representation of the data. Adapting or replacing these applications would require a prohibitive migration effort. Therefore, a mediation approach is needed that bridges the conceptual gap between the relational model and RDF, resulting in a cooperative use of the data in RDF-based as well as relational applications.

In the past, various RDB-to-RDF mediation approaches were explored, resulting in the definition of multiple RDB-to-RDF mappings and algorithms to translate Semantic Web queries to the RDB. However, all of these approaches are limited to read-only data access and have a strong focus on SPARQL for querying and Linked Data for browsing the data as RDF. Use cases where write access to the RDB or support for other data access approaches is needed have so far been neglected by the state-of-the-art RDB-to-RDF mediation approaches.

In this dissertation we present OntoAccess, an RDB-to-RDF mediation approach that enables RDF-based read and write access to an RDB. The approach consists of three parts: (1) the RDB-to-RDF mapping called R3M that provides the basis for RDF-based read and write access to the RDB; (2) algorithms to translate RDF-based read and write requests to the RDB; and (3) an architecture for an extensible RDB-to-RDF mediation that enables support for multiple data access approaches.

To validate our OntoAccess approach for RDB-to-RDF mediation we provide the following: (1) a formal definition of our RDB-to-RDF mapping R3M and proofs of its bidirectional properties; (2) a performance evaluation of our algorithms for translating RDF-based requests to the RDB; (3) a proof of concept implementation of our architecture for an extensible RDB-to-RDF mediation platform; and (4) a case study in the domain of software analysis where we apply OntoAccess to make a data bridge between an RDB-based legacy system and its RDF-based long-term replacement.

In summary, we therefore state: *The* OntoAccess *approach, consisting of a mapping, an architecture, and algorithms, bridges the conceptual gap between the relational data model and RDF and therefore enables RDF-based read and write access to an RDB.*

Contents

List of Figures

List of Tables

List of Listings

1

Synopsis

Relational Databases (RDBs) are used in most current enterprise environments to store and manage data. According to [Chang et al., 2004], the majority of structured data on the Web is relational. While RDBs are well suited to handle large amounts of data, they were not designed to preserve the data semantics. The meaning of the data is implicit at the application level but not explicitly encoded in the relational model.

The Semantic Web provides a common framework that allows data to be shared and reused across application, enterprise, and community boundaries [World Wide Web Consortium, 2011]. Although developed for the Web, these Semantic Web technologies have proven to be useful in other domains as well, especially if data from different sources has to be exchanged or integrated (*e.g.*, [Patel et al., 2009, Ma et al., 2009, Langegger et al., 2008, Hert et al., 2011a]).

In existing systems, however, it is not always possible or desirable to convert all relational data to RDF as other business-critical applications rely on the *relational* representation of the data. Adapting or replacing these applica-

tions would require a prohibitive migration effort. Therefore, an RDB-to-RDF mediation approach is needed that bridges the conceptual gap between the relational model and RDF (*i.e.*, RDF requests are translated on demand to the RDB using an RDB-to-RDF mapping). The result is a cooperative use of the data in RDF-based as well as relational applications. In addition, mediation allows one to further exploit the advantages of the well established database technology such as query performance, scalability, transaction support, and security.

This synopsis is organized as follows. In Section 1.1, we present the motivation of this dissertation by highlighting the two major limitations of the state-of-the-art RDB-to-RDF mediation approaches. Section 1.2 provides a brief overview of our RDB-to-RDF mediation approach called OntoAccess. The overall research goal and the research questions addressed in this dissertation are discussed in detail in Section 1.3. Section 1.4 describes the publications that provide the foundation of this dissertation, including links to the research questions. Section 1.5 enumerates the main contributions of this dissertation while Section 1.6 discusses the limitations of our approach and possible directions for future research. Section 1.7 presents a roadmap for the remainder of this dissertation.

1.1 Motivation

Mapping RDBs to RDF is an active field of research. Bridging the conceptual gap between RDF and the relational model requires a mapping from concepts in an RDB schema to terms defined in an RDF vocabulary or ontology. In the past, various RDB-to-RDF approaches were explored, resulting in the definition of multiple RDB-to-RDF mapping languages. Several of these mapping languages were tailored to specific use cases and application scenarios. They range from simple representations of the RDB schema in RDF to complex mappings and transformation of an RDB schema to an ontology. Only few were developed as general-purpose RDB-to-RDF mapping languages. However, all of these existing mapping languages are limited to read-only data access. Use cases where write access to the RDB is needed have so far been neglected.

As a result, these approaches are limited to data warehouse-like applications where the data can be queried and analyzed but not modified. We therefore identify the lack of support for RDF-based write access to an RDB as the first major limitation of the state-of-the-art RDB-to-RDF mediation approaches.

Several approaches exist to access RDF data. They range from recommendations of the World Wide Web Consortium[1] (W3C) such as SPARQL [Prud'hommeaux and Seaborne, 2008] to community-driven solutions such as Semantic Web frameworks (*e.g.*, Jena,[2] Sesame,[3] RDF2Go[4]) and best practices such as Linked Data[5] for browsing RDF data. SPARQL as the standard RDF query language and Linked Data currently represent the most widely adopted approaches for RDF data access on the Semantic Web. As a result, existing RDB-to-RDF mediators have a strong focus on supporting those two data access approaches. This explains, at least to a certain degree, their limitation to read-only data access because SPARQL (as defined in [Prud'hommeaux and Seaborne, 2008]) as well as Linked Data are read-only. Other approaches for accessing RDF data, including those that support write access, cannot be used with the existing RDB-to-RDF approaches. We therefore identify the lack of extensibility to support multiple data access approaches as the second major limitation of the state-of-the-art RDB-to-RDF mediation approaches.

In this dissertation, we address the two major limitations highlighted in this section in the OntoAccess approach. First, we show the need for RDF-based write access to RDBs and present an RDB-to-RDF mediation approach that supports RDF-based read and write access to the RDB. We call this *bidirectional* RDB-to-RDF mediation. We introduce the bidirectional RDB-to-RDF mapping *R3M* as well as algorithms to translate RDF-based write requests to the RDB, *i.e.*, to SQL data manipulation language (DML) statements. Second, we provide an architecture for an RDB-to-RDF mediation platform that is extensible *w.r.t.* data access approaches. We further describe our proof of concept implementation of the mapping, the translation algorithms, and the mediation platform.

[1] http://w3c.org/
[2] http://openjena.org/
[3] http://www.openrdf.org/
[4] http://rdf2go.semweb4j.org/
[5] http://linkeddata.org/

To summarize, we state: *The* ONTOACCESS *approach, consisting of a mapping, an architecture, and algorithms, bridges the conceptual gap between the relational data model and RDF and therefore enables RDF-based read and write access to an RDB.*

1.2 OntoAccess in a Nutshell

ONTOACCESS is an RDB-to-RDF mediation approach that enables RDF-based read and write access to RDBs. It consists of the bidirectional RDB-to-RDF mapping called R3M and an extensible platform for RDB-to-RDF mediation that implements algorithms to translate RDF-based read and write requests to the RDB.

R3M implements a direct mapping from an RDB schema to an RDF vocabulary based on the approach described in [Berners-Lee, 2009b]. It provides several extensions to this approach to hide particularities of the relational model that are not needed in RDF (*e.g.*, link tables). The mapping is explicit, which means it is not based on SQL queries (or views) defined by the mapping author, but provides explicit constructs for mapping database tables and attributes to RDF vocabulary terms. This is needed to enable write access for all valid mappings. In contrast, mapping languages that rely on SQL are, in general, affected by the view update problem [Bancilhon and Spyratos, 1981] and are therefore impractical if write access is required (*cf.* [Garrote and Garcia, 2011]). Listing 1.1 depicts examples for the three main mapping elements of R3M: *TableMap*, *AttributeMap*, and *LinkTableMap*.

Listing 1.1a) describes a *TableMap* representing the mapping of a database table to a class of the RDF vocabulary. It contains the name of the table (line 2) and the class it is mapped to (line 3). The URI pattern (line 4) is used to generate the URIs for instances of this table, based on values of table attributes that are specified between double percentage signs (*e.g.*, %%*id*%% where *id* is the name of the primary key attribute). A *TableMap* further contains a list of *AttributeMap*s (lines 5 to 6). Listing 1.1b) presents an example of an *AttributeMap* that maps a database attribute to a property of the RDF vocabulary. It contains the name of the attribute in the database schema (line 9)

```
1  a) ex:author  a  r3m:TableMap;
2       r3m:hasTableName  "author";
3       r3m:mapsToClass   foaf:Person;
4       r3m:uriPattern    "http://example.org/author%%id%%";
5       r3m:hasAttribute  ex:author_id, ex:author_email,
6                         ex:author_firstname, ex:author_lastname.
7
8  b) ex:author_email  a  r3m:AttributeMap;
9       r3m:hasAttributeName      "email" ;
10      r3m:mapsToObjectProperty  foaf:mbox;
11      r3m:hasConstraint         [ a r3m:NotNull ].
12
13 c) ex:publication_author  a  r3m:LinkTableMap;
14      r3m:hasTableName          "publication_author";
15      r3m:mapsToObjectProperty  dc:creator;
16      r3m:hasSubjectAttribute   ex:pa_publication;
17      r3m:hasObjectAttribute    ex:pa_author.
```

Listing 1.1: Example R3M Mapping Elements

and the property it is mapped to (line 10). Additionally, an *AttributeMap* includes information about constraints defined on that attribute (*e.g.*, a *not null* constraint; line 11). Listing 1.1c) shows a *LinkTableMap* representing the mapping of a link table to a property of the RDF vocabulary. It specifies the name of the link table in the database (line 14) and the property it is mapped to (line 12). A link table always contains two foreign key attributes that point to the tables of the N:M relationship. These attributes are represented as *AttributeMap*s (line 16 and 17; not shown in the example) that provide the names of the attributes, the foreign key references to the tables, and the direction of the relationship (from subject to object).

ONTOACCESS is designed as an extensible platform for bidirectional RDB-to-RDF mediation. It is based on mappings in R3M and algorithms for RDF-based read and write access to the RDB. In addition, it provides an extensible set of data access interfaces, namely for SPARQL [Prud'hommeaux and Seaborne, 2008] (including SPARQL/Update [Seaborne et al., 2008]), Linked Data, Semantic Web frameworks (Jena, Sesame, and RDF2Go), and Change-

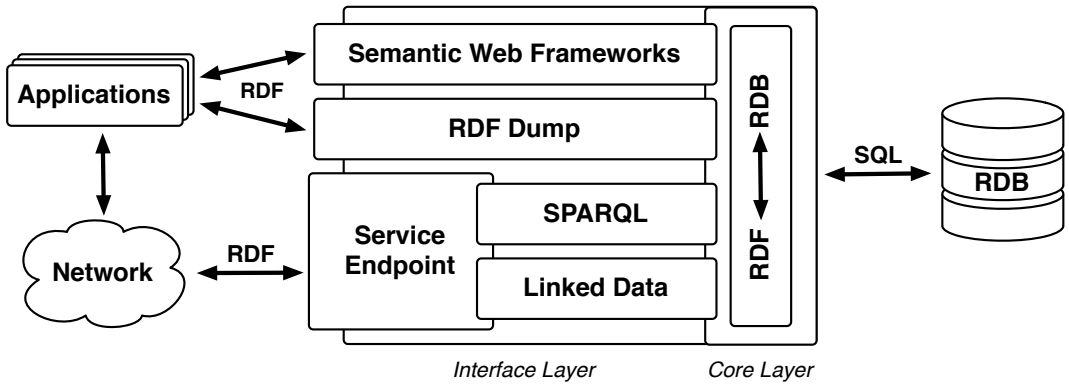

Figure 1.1: OntoAccess Platform Architecture

Set.[6] Figure 1.1 presents an overview of the two-layer architecture of the ONTOACCESS platform. The lower part, called core layer, is responsible for the actual RDB-to-RDF translation as well as the interaction with the database system. The upper part, called interface layer, exposes the functionality of the ONTOACCESS core to the individual data access approaches. The interfaces are either accessed directly by applications (*e.g.*, Semantic Web Frameworks, RDF Dump) or over the network via a service endpoint (*e.g.*, SPARQL, Linked Data).

The conceptual gap between the relational model and RDF has a greater effect on translating RDF-based write requests to the RDB than on read-only queries. In contrast to an invalid read request that results in an empty response, an invalid write request cannot be fully processed and results in an error. A write request is considered invalid if it refers to ontology terms or instances that cannot be mapped to the RDB or if it violates constraints defined in the database schema (*e.g.*, a *not null* constraint). In ONTOACCESS, this problem is alleviated with a semantic feedback protocol. The main idea of this feedback approach is to detect invalid requests already during translation and to provide feedback in a semantic format such as RDF. In this way, a conceptual break between request and (error) response can be avoided.

[6] `http://docs.api.talis.com/getting-started/changesets`

In summary, the OntoAccess approach enables RDF-based read and write access to an RDB by bridging the conceptual gap between the relational model and RDF.

1.3 Research Goal & Questions

Our overall research goal was to develop the OntoAccess approach consisting of a mapping, an architecture, and algorithms that enables RDF-based read and write access to an RDB.

Based on this overall research goal, this dissertation addresses six research questions.

The first two research questions concern the mapping from an RDB schema to an RDF vocabulary or ontology.

Research Question 1 (RQ1): *What properties does an RDB-to-RDF mapping need to have to enable RDF-based read and write access to an RDB?*

RQ1 addresses the definition of an RDB-to-RDF mapping that enables RDF-based read and write access to an RDB. Existing mapping approaches are limited to read-only use cases and data access (*cf.* [Hert et al., 2011c]). This allows them to rely on SQL for the definition and implementation of the mapping. However, such approaches are, in general, impractical if RDF-based *write* access to an RDB should be enabled, due to the view update problem known from database research. The view update problem [Bancilhon and Spyratos, 1981] states that updates performed on SQL views can not uniquely be propagated to updates on the base tables for all but the most trivial view definitions. To ensure that all defined mappings support write access, SQL can not be used. An explicit mapping needs to be defined with properties that take the requirements of write access into consideration. A formal definition of the mapping is used to prove that this mapping has the required properties to enable RDF-based read and write access to an RDB.

Research Question 2 (RQ2): *What level of expressivity does a mapping need to cover real world application scenarios?*

RQ2 addresses the expressivity of an RDB-to-RDF mapping and its applicability in real world scenarios, *i.e.*, RDB schemata and use cases that are not specifically designed for RDB-to-RDF mapping. A mapping defined as described in *RQ1* will inherently be less expressive *w.r.t.* mapping features than existing, read-only languages that rely on SQL. We investigate what level of expressivity and therefore what mapping features are needed to apply the mapping in real world scenarios. A feature-based comparison of RDB-to-RDF mapping languages and a case study are used to show the applicability of our RDB-to-RDF mapping in real world scenarios.

The next two research questions concern algorithms for the translation of RDF-based read and write requests to the RDB.

Research Question 3 (RQ3): *What algorithms are needed to translate the RDF CRUD[7] operations to the RDB while providing comparable performance to existing, read-only RDB-to-RDF approaches and RDF triple stores?*

RQ3 addresses the development of algorithms to translate the RDF CRUD operations to the RDB. CRUD operations are basic operations to work with data. New data is *Create*d and existing data is *Read*, *Update*d, or *Delete*d. In RDF, the CRUD operations translate to the following: *Create* adds RDF triples to the data set while *Delete* removes triples. *Update* can be interpreted as an atomic combination of *Delete* and *Create* and is therefore not addressed separately. *Read* is mapped to querying for RDF triples based on a triple pattern where subject, predicate, and object can each be a variable that is matched against the data set. To apply RDB-to-RDF mediation in practice, the performance of RDF-based read and write access should be comparable to existing, read-only RDB-to-RDF approaches and RDF triple stores. Therefore, algorithms to translate RDF CRUD operations to the RDB are needed that are efficient *w.r.t.* execution

[7] **C**reate, **R**ead, **U**pdate, **D**elete

times. A proof of concept implementation of these algorithms is used to conduct a performance evaluation that compares the algorithms to state-of-the-art RDB-to-RDF approaches and RDF triple stores.

Research Question 4 (RQ4): *What algorithms are needed to translate SPARQL/Update requests to the RDB?*

RQ4 addresses the development of algorithms to translate SPARQL/Update [Seaborne et al., 2008] requests to the RDB. SPARQL/Update was proposed to the W3C as a data manipulation language (DML) for RDF data. It consists of operations to add (`INSERT DATA`) and to delete (`DELETE DATA`) RDF triples as well as to update (`MODIFY`) an RDF graph based on triple pattern matching known from the SPARQL query language. The SPARQL/Update proposal was selected as the basis for the official W3C recommendation for an RDF DML [Schenk et al., 2010]. It is expected to be an important approach for write access to RDF data. An RDB-to-RDF mediator should provide support for SPARQL/Update. Therefore, algorithms to translate SPARQL/Update operations to the RDB are needed. The feasibility of these algorithms is shown with a proof of concept implementation.

The remaining two research questions concern the architecture of the RDB-to-RDF mediator.

Research Question 5 (RQ5): *What characteristics does an architecture of an RDB-to-RDF mediation platform need to support an extensible set of data access approaches?*

RQ5 addresses the architecture of an RDB-to-RDF mediation platform that provides support for multiple data access approaches and that is extensible to add support for additional ones without the need to re-implement the core RDB-to-RDF translation logic. Although SPARQL, as the W3C recommendation of a standard query language for RDF data, is widely used, other data access approaches such as Linked Data or Semantic Web frameworks (*e.g.*, Jena, Sesame, RDF2Go) exist and are

applied in practice. Work on data access approaches is not completed as new approaches are still being developed and standardized. For instance, the SPARQL 1.1 Working Group[8] introduced a new data access approach in the form of the *Graph Store HTTP Protocol* [Ogbuji, 2011], targeted at cases where SPARQL is not a viable solution because of its complexity. An RDB-to-RDF mediator should provide support for an extensible set of data access interfaces without the need to re-implement the main RDB-to-RDF translation logic for each interface. An architecture is needed that supports this extensibility of data access approaches. The feasibility of such an architecture is shown with a proof of concept implementation that includes an extensive set of data access interfaces.

Research Question 6 (RQ6): *What kind of semantic feedback should be provided to bridge the conceptual gap between the relational model and RDF?*

RQ6 addresses the conceptual gap between the relational model and RDF and how it can be bridged in the case of invalid write requests. This conceptual gap between the relational model and RDF affects the translation of RDF-based requests to the RDB, especially in the case of write requests. In contrast to an invalid read request that results in an empty response, an invalid write request cannot be fully processed and results in an error. A write request is considered invalid if it refers to ontology terms or instances that cannot be mapped to the RDB or if it violates constraints defined in the database schema (*e.g.*, *not null* constraints). Such errors should be reported in a semantic format such as RDF to avoid a conceptual break between request and (error) response. The feasibility of such a semantic feedback protocol is shown with a proof of concept implementation.

The relations between the research questions and the publications that constitute the foundation of this dissertation are illustrated in Figure 1.2. It consists of three dimensions that reflect the three groups of research questions presented in this section, namely *Mapping*, *Algorithms*, and *Architecture*. The

[8] `http://www.w3.org/2001/sw/DataAccess/`

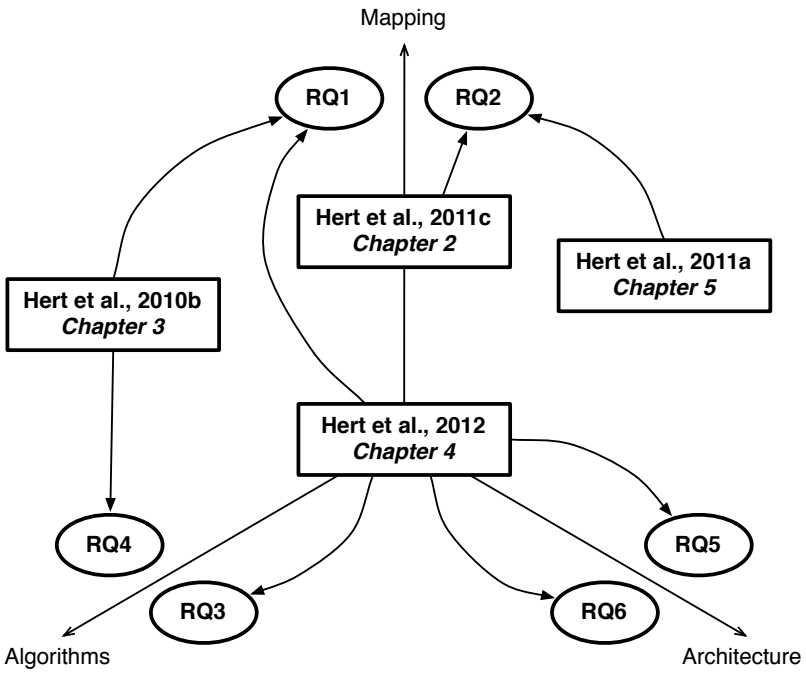

Figure 1.2: Overview of Relations between Research Questions and Publications

research questions are represented as circles and the publications as rectangles. An arrow from a publication to a research question implies that this publication addresses that research question. Section 1.4 summarizes each publication and describes how the research questions are addressed.

1.4 Foundation of the Dissertation

The foundation of this dissertation is a set of selected publications that were published in international, peer-reviewed venues in the area of Semantic Web and database research. Appendix A provides a list of all publications that are related to this dissertation.

Chapter 2 In [Hert et al., 2011c], we present related work in RDB-to-RDF mapping languages. Several approaches were explored to make relational data available to Semantic Web-enabled applications. Depending

on the requirements, these approaches introduced mapping languages that range from simple and pragmatic to highly specific or general-purpose. This development lead to the standardization effort of the World Wide Web Consortium (W3C) carried out in the RDB2RDF Working Group[9] (WG). The mission of the RDB2RDF WG as defined in their charter [Halpin and Herman, 2009] is *to standardize a language for mapping relational data and relational database schemas into RDF and OWL*. Although a standard mapping language is currently under development, we argue that alternative languages still have a right to exist as they may provide features or simplicity required in certain use cases that the standard mapping language cannot provide or explicitly excludes.

The goal and contribution of this paper is to provide a feature-based comparison of the state-of-the-art RDB-to-RDF mapping languages. It should act as a guide in selecting a mapping language for a given application scenario and its requirements *w.r.t.* mapping features. We develop a comparison framework based on fifteen features extracted from the document *Use Cases and Requirements for Mapping Relational Databases to RDF* [Prud'hommeaux and Hausenblas, 2010] developed by the RDB2RDF WG as well as a literature survey on existing RDB-to-RDF mapping languages. We apply this comparison framework to nine state-of-the-art RDB-to-RDF mapping languages. As a result, we propose a classification into four categories of mapping languages and provide recommendation for selecting a mapping language. The four categories are:

Direct Mapping: The direct mapping, as its name implies, is a direct approach for RDB-to-RDF mapping. Aimed at providing simple means to map RDBs to RDF, it does not hide the particularities of the relational model such as link tables in the RDF representation. Such helper constructs are not required in RDF as it provides direct means to model these relationships. The advantage of this mapping

[9] `http://www.w3.org/2001/sw/rdb2rdf/`

is its simplicity to understand and implement the language. It is therefore recommended in application scenarios where a direct representation of the relational schema is acceptable and simplicity is of higher value.

Read-only General-purpose Mapping: Read-only general-purpose mapping languages enable a highly expressive bridging of the conceptual gap between RDF and the relational model. However, this higher expressivity also implies an increased complexity that results in an unidirectional mapping, *i.e.*, bidirectional read and write access to the data is impractical. Understanding and implementing the mapping languages may also require a higher learning effort. Due to their high expressivity, mapping languages of this category are recommended for various application scenarios as long as the usage is limited to read-only data access.

Read-Write General-purpose Mapping: Read-Write general-purpose mapping languages provide a more expressive bridging of the conceptual gap between the relational model and RDF than the mapping approaches of the direct mapping. Particularities of the relational model such as link tables are not transferred to the RDF representation. These mapping languages are, however, less expressive than the read-only general-purpose mapping languages, but this is required to ensure support for write access. Mapping languages of this category are recommended for application scenarios where RDF-based read and write access to the relational data is needed.

Special-purpose Mapping: Special-purpose mapping languages were developed for specific use cases and are obviously influenced by the requirements of those use cases in the features they support. This does not necessarily result in a loss of expressivity or applicability compared to the general-purpose languages. As the mapping languages of this category were developed for their specific application scenarios, they are recommended for application in closely related

scenarios where the general-purpose mapping languages are not applicable or too complex.

The feature-based comparison of RDB-to-RDF mapping languages presented in this paper addresses *Research Question 2*. The comparison includes our own mapping language R3M and therefore investigates the differences in feature support and implicitly the expressivity of R3M and the state-of-the-art RDB-to-RDF mapping languages.

Chapter 3 In [Hert et al., 2010b], we present results on enabling RDF-based write access to RDBs. Existing approaches for mapping RDBs to RDF focus on *exposing* the relational data to the Semantic Web. They provide SPARQL endpoints to query the data, but they do not address data updates. Our contribution in this paper is the RDF-based write access to relational data via SPARQL/Update. We introduce our RDB-to-RDF mapping called R3M and we present algorithms for translating SPARQL/Update requests to SQL DML.

R3M is an update-aware RDB-to-RDF mapping that records additional information about the database schema to support data manipulations and to detect invalid update requests during the translation process. Updatability and simplicity were two of the main design goals of this mapping. It is expressed in RDF and models the mappings between terms of a domain ontology and the database schema and it records additional information about the schema and its integrity constraints. The mapping employs the approach where database tables are mapped to ontology classes and attributes to properties (*cf.* [Berners-Lee, 2009b]). This means that each database table representing a concept in the application domain is mapped to an ontology class representing the same concept. Likewise, each database attribute that constitutes a relationship between an entity and a data value (or another entity) is mapped to an ontology property that links instances of a class to literal values (or other instances). Thereby, each row in a database table is mapped to a set of RDF triples. One triple identifies the entity that is represented by this row

as an instance of the class the corresponding table is mapped to. Then, there exists in principle one triple for each table attribute that relates the instance to a data value or another instance (*e.g.*, foreign keys). Link tables are used in RDBs to describe N:M relationships among relations. In RDF, such auxiliary constructs are not needed, which is why R3M features explicit support to map these tables to object properties instead of classes.

SPARQL/Update as defined in [Seaborne et al., 2008] consists of three main request types: `INSERT DATA`, `DELETE DATA`, and `MODIFY`. The first two request types are basically a list of RDF triples to either add to or remove from the data set. A `MODIFY` request is similar to a SPARQL `CONSTRUCT` query but it results in two newly constructed RDF graphs, whereof one contains triples to remove from and one contains triples to add to the data set. In short, a `MODIFY` request can be evaluated as a SPARQL query and then broken down into one `INSERT DATA` and one `DELETE DATA` request. The translation for adding and removing triples is basically the same. The difference is in the generated SQL statements (insert vs. delete). First, the triples are grouped into so-called subject groups based on equal subjects (*i.e.*, these triples have the same subject and therefore affect the same record in the database). This allows for an individual translation of each group of triples. Second, the affected table is identified via the subject URI. In a third step, the mapping is used to check if the submitted triples satisfy certain integrity constraints of the database schema. Step four generates the SQL DML statement for adding or deleting this group of triples based on the mapping definition. The predicate of each triple is translated to an attribute of the respective table. The object is used as the data value either directly, if it is a literal, or by matching it against a template in the mapping and extracting a predefined substring, if it is a resource URI. Each subject group is processed that way and the resulting SQL statements are collected. Finally, the statements are executed within a single database transaction to ensure the atomicity of the original request.

We further present a feasibility study of our prototype implementation based on a synthetic data set from the Berlin SPARQL Benchmark [Bizer and Schultz, 2009].

This paper addresses *Research Question 1* by introducing our RDB-to-RDF mapping R3M that enables RDF-based read and write access to the RDB. Further, *Research Question 4* is addressed by presenting algorithms for translating SPARQL/Update requests to the SQL DML.

Chapter 4 In [Hert et al., 2012], we present OntoAccess as an extensible platform for RDF-based read and write access to existing data stored in RDBs. We state that there exist many different data access approaches in current Semantic Web applications such as SPARQL, Semantic Web frameworks (*e.g.*, Jena, Sesame, RDF2Go), and Linked Data, but also upcoming approaches such as SPARQL/Update and the SPARQL 1.1 Graph Store HTTP Protocol. We argue that a platform-based approach is needed to avoid repeated implementation effort in RDB-to-RDF translation. We identify three basic CRUD operations that such a platform has to provide in its core implementation, namely (1) querying for a single triple pattern, (2) adding a set of triples, and (3) removing a set of triples. These basic operations are implemented in the core of OntoAccess and we show that this architectural decision enables the simple implementation of various data access interfaces.

We show that this platform-based approach performs comparably to existing read-only RDB-to-RDF mapping approaches (*i.e.*, D2R [Bizer and Cyganiak, 2006]) and current RDF triple stores (*i.e.*, Jena SDB,[10] Jena TDB[11]).

The conceptual gap between the relational model and RDF has a greater effect on translating RDF-based write requests to the RDB than on read-only queries. If a query uses ontology terms (or instances) that cannot be mapped to the database schema (or data), the query can be processed

[10] `http://openjena.org/SDB/`
[11] `http://openjena.org/TDB/`

without error but simply returns no results. However, if a write request contains such non-mappable terms it cannot be fully processed and results in an error. Further, a write request can be invalid if it violates constraints defined in the database schema (*e.g.*, *not null* constraints). In such situations, it is possible to simply reject the request or ignore the parts that cannot be mapped. In both approaches, the client does not know why the request was not fully processed, especially if the client is unaware of the RDB-based data storage. Having said that, it cannot be expected (neither is it desirable) that all clients know about the specifics of the RDB schema if their usage is limited to the RDF-based data access interfaces. We therefore propose a semantic feedback protocol to alleviate this problem. The main idea of this feedback approach is to detect requests that are invalid *w.r.t.* the RDB schema already during request translation and in a second step provide feedback to the client in a semantic format such as RDF. In this way, a conceptual break between request and (error) response can be avoided.

We further introduce a formal definition of our RDB-to-RDF mapping as well as proofs of its bidirectional properties. The rationale of the formal definition and the proofs is to show that our mapping R3M enables RDF-based read and write access to an RDB without being affected by the view update problem.

The architecture for an extensible RDB-to-RDF mediation platform presented in this paper addresses *Research Question 5*. The algorithms for translating RDF CRUD operations to the RDB as well as the performance evaluation address *Research Question 3*. Further, *Research Question 1* is addressed by formalizing our RDB-to-RDF mapping and by providing proofs of its bidirectional properties. The semantic feedback protocol introduced in this paper addresses *Research Question 6*.

Chapter 5 [Hert et al., 2011a] presents a case study in the software analysis domain on how ONTOACCESS can be used to facilitate the transition from an RDB-based legacy system to a Semantic Web-enabled system in prac-

tice. The case study shows how we successfully applied OntoAccess to advance our Eclipse-based software evolution analysis framework Evolizer [Gall et al., 2009] to Sofas [Ghezzi and Gall, 2011], a service-oriented, distributed, and collaborative software analysis platform.

To motivate our case study, we present use cases that require interoperability between Evolizer and Sofas. These use cases need a bidirectional data exchange, *i.e.*, from Evolizer to Sofas and vice versa. First, Evolizer contains data about the software life-cycle of hundreds of software systems. Re-importing this vast amount of data in Sofas from version control and bug tracking systems would take months, and some of these repositories might not even be available online anymore. Therefore, RDF-based *read access* to the Evolizer database is needed. Second, Evolizer implements importers to import source code and history data from centralized version control systems, such as CVS and SVN. Lately decentralized version control systems, such as Git or Mercurial, have gained popularity. Therefore, respective import services were developed for the Sofas platform. The data produced by these importer services is modeled in RDF. It would also be valuable to Evolizer because existing tools could be used to leverage it. This, however, requires RDF-based *write access* to the Evolizer database. Lastly, Sofas implements an extensible framework to compute software metrics on the data. Again, this data is modeled in RDF, but matching relations are available in the Evolizer database schema. RDF-based *write access* to the RDB is needed to make the metrics data available to Evolizer. These use cases indicate that, for making a bridge between Evolizer and Sofas, an RDB-to-RDF mapper such as OntoAccess is needed that provides RDF-based read and write access to RDBs.

During this case study, we faced several challenges *w.r.t.* the mapping in OntoAccess. The first challenge is related to the representation of concept inheritance in RDB systems. Inheritance is a central concept in the object-oriented methodology and is therefore commonly used in object-oriented systems, including Evolizer. Relational databases, un-

like object-relational or object-oriented databases, do not directly support inheritance. However, there exist principal strategies to implement inheritance in relational database schemata (*cf.* [Garcia-Molina et al., 2008]). We describe extensions to our mapping language that are needed to support these strategies. The second challenge is related to defining the RDB-to-RDF mappings. Mappings in OntoAccess are encoded in RDF, which makes them well-suited for automatic processing by machines but hinders the accessibility for human users. Manually defining such mappings is a time-consuming and error-prone task, mostly consisting of repetitive steps. Therefore, tool support for defining mappings is indispensable in more complex application scenarios where the number of database tables and columns is of significance. We built a tool [Brügger, 2009] to ease the definition of OntoAccess mappings. It semi-automatically generates a mapping from an RDB schema in two steps. First, it automatically generates a basic mapping, based on information extracted from the schema catalog of the database system. Terms of the target ontology are also generated in this step, based on table and column names in the database schema. Next, the tool displays a graphical editor for refining the mapping. This step is mainly concerned with replacing the generated terms with actual terms from the target ontology. The tool further provides validation of existing mappings to catch errors from manual editing. The tool is implemented as a plug-in for the ontology editor Protégé[12] to enable quick access to the definition of the target ontology.

In summary, judging from the experiences made in our case study, we are confident that OntoAccess is a valuable tool that will foster the acceptance of Semantic Web technology in practice.

The case study presented in this paper addresses *Research Question 2*. The RDB schema and the target ontology of this case study were not developed for RDB-to-RDF mapping but represent a real world application scenario.

[12] `http://protege.stanford.edu/`

1.5 Contributions

The overall contribution of this dissertation is our approach OntoAccess to enable RDF-based read and write access to an RDB.

In summary, the main contributions of this dissertation are:

1. The bidirectional RDB-to-RDF mapping *R3M* that enables mappings defined in this language to be used for RDF-based read and write access to an RDB.
2. A set of algorithms for translating SPARQL/Update requests to the RDB.
3. A set of algorithms for translating RDF CRUD operations to the RDB.
4. An architecture for an extensible RDB-to-RDF mediation platform that supports multiple data access interfaces without the need for re-implementing the main RDB-to-RDF translation logic.
5. A proof of concept implementation of OntoAccess as an extensible RDB-to-RDF mediation platform. Its performance is comparable to existing, read-only RDB-to-RDF mediation approaches and RDF triple stores.
6. A semantic feedback protocol to bridge the conceptual gap between the relational model and RDF.
7. A framework that allows for a comparison of RDB-to-RDF mapping languages on a feature-by-feature basis.
8. A feature-based comparison of the state-of-the-art RDB-to-RDF mapping languages.

1.6 Limitations and Future Work

The OntoAccess approach and implementation as described in this dissertation has its limitations. Several of these limitations provide opportunities for future research directions. The main limitations are:

- The expressivity of our RDB-to-RDF mapping R3M is lower than some of the existing, read-only mapping languages (*cf.* [Hert et al., 2011c]) due to the additional requirement of enabling RDF-based write access to the RDB. However, we did not formally investigate if the current set of features supported in R3M provides the maximal expressivity for a bidirectional RDB-to-RDF mapping or if additional features could be supported without loosing the possibility of RDF-based write access to the RDB. A future research direction would be to investigate R3M in this respect.

- Besides the mapping between database elements and RDF vocabulary terms, our RDB-to-RDF mapping language R3M stores additional information about the RDB schema in the mapping definition, namely information about datatypes, primary and foreign keys as well as (integrity) constraints. Currently, the support in R3M and our prototype implementation is limited to *not null*, *default*, *unique*, and *check* constraints. The dynamics introduced by newer functionalities such as *triggers* are not considered but would be an interesting direction for future research.

- OntoAccess is limited to one single RDB as a data source. It does not incorporate any query or update federation. It would be an interesting direction for future research to extend OntoAccess in this respect and to apply state-of-the-art federation techniques on top of it. The explicit information provided in the mapping about what kind of data (*i.e.*, classes and properties of an RDF vocabulary) are stored in each data source could be leveraged to make the federation more efficient.

- The prototype implementation of OntoAccess does not support reasoning. It was shown in [Calvanese et al., 2007b] that reasoning for the DL-Lite [Calvanese et al., 2007b] family of description logics (and therefore for the OWL 2 QL profile [Motik et al., 2009]) is possible by (relational) query rewriting. It would therefore be feasible to add reasoning support to an RDB-to-RDF mediation system such as OntoAccess.

- In this dissertation, we introduced a semantic feedback protocol to bridge the conceptual gap between the relational model and RDF in case of invalid write requests. We provided a proof of concept implementation, but a thorough evaluation is needed to show the usability of this feedback approach.

- When we started our work on RDB-to-RDF mediation in 2008, RDF triple stores were unable to efficiently handle large amounts of data, especially if compared to state-of-the-art RDB systems (*cf.* [Bizer and Schultz, 2008a]). In the meantime, the performance of RDF triple stores has improved so that state-of-the-art RDF triple stores are able to handle data sets of significant size.[13] This provides the opportunity to address the RDB-to-RDF mediation problem from the opposite angle. Instead of leaving the data in the RDB and translating Semantic Web requests to SQL statements, all data could be migrated to RDF and SQL requests of the legacy applications could be translated to Semantic Web requests (*e.g.*, SPARQL). With this approach, the data would already be migrated and the transition to Semantic Web technologies would be complete as soon as the last legacy application is adapted, replaced, or decommissioned.

1.7 Roadmap

The roadmap for the remainder of this dissertation is depicted in Figure 1.3. It lists each of the remaining chapters and the details of the corresponding publications. The publications are sorted in the intended order of reading.

Chapter 2 (p.25) provides an overview of related work in RDB-to-RDF mapping and a feature-based comparison of the mapping languages.

Chapter 3 (p.45) introduces our RDB-to-RDF mapping R3M by example and describes algorithms for translating SPARQL/Update requests to the RDB.

[13] *e.g.*, see recent results of the Berlin SPARQL Benchmark at `http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/results/V6/`

Chapter 4 (p.69) presents our architecture for an extensible RDB-to-RDF mediation platform and algorithms for translating basic RDF CRUD operations to the RDB. It further provides a formal definition of our RDB-to-RDF mapping R3M and a performance evaluation.

Chapter 5 (p.103) describes a case study where we applied ONTOACCESS in the domain of software analysis.

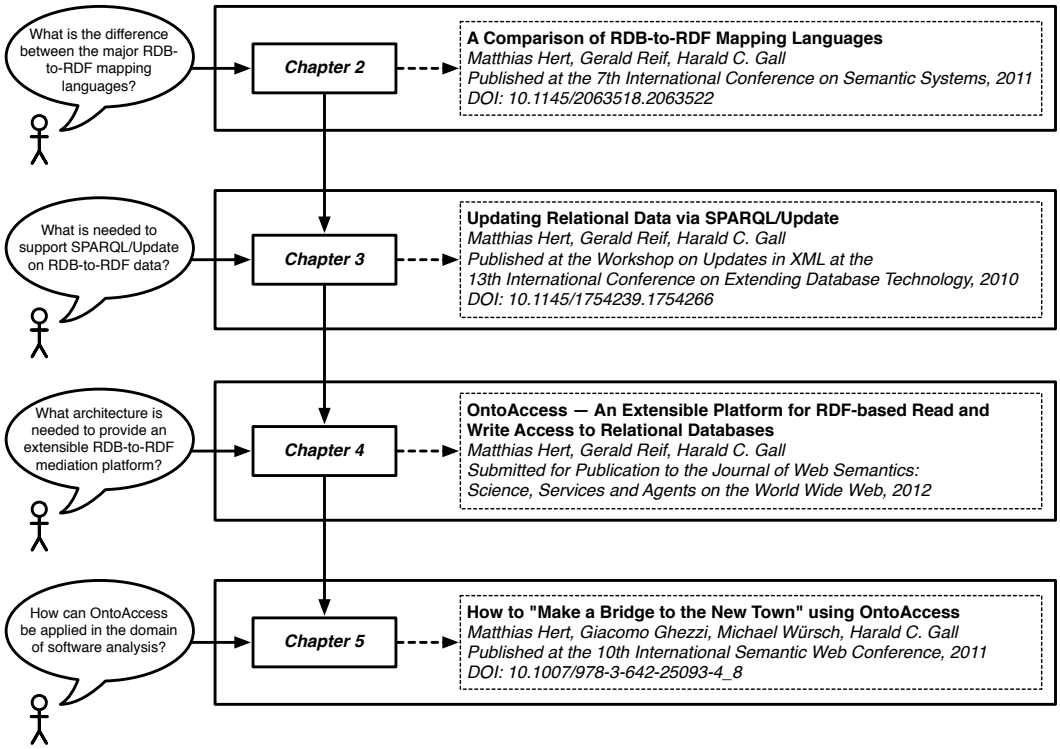

Figure 1.3: Dissertation Roadmap

2

A Comparison of RDB-to-RDF Mapping Languages

Matthias Hert, Gerald Reif, Harald C. Gall
Published at the 7th International Conference on Semantic Systems, 2011
DOI: 10.1145/2063518.2063522

Abstract

Mapping Relational Databases (RDB) to RDF is an active field of research. The majority of data on the current Web is stored in RDBs. Therefore, bridging the conceptual gap between the relational model and RDF is needed to make the data available on the Semantic Web. In addition, recent research has shown that Semantic Web technologies are useful beyond the Web, especially if data from different sources has to be exchanged or integrated. Many mapping languages and approaches were explored leading to the ongoing standardization effort of

the World Wide Web Consortium (W3C) carried out in the RDB2RDF Working Group (WG). The goal and contribution of this paper is to provide a feature-based comparison of the state-of-the-art RDB-to-RDF mapping languages. It should act as a guide in selecting an RDB-to-RDF mapping language for a given application scenario and its requirements *w.r.t.* mapping features. Our comparison framework is based on use cases and requirements for mapping RDBs to RDF as identified by the RDB2RDF WG. We apply this comparison framework to the state-of-the-art RDB-to-RDF mapping languages and report the findings in this paper. As a result, our classification proposes four categories of mapping languages: direct mapping, read-only general-purpose mapping, read-write general-purpose mapping, and special-purpose mapping. We further provide recommendations for selecting a mapping language.

2.1 Introduction

Mapping Relational Databases (RDB) to RDF is an active field of research. As reported in [Chang et al., 2004], the majority of data on the current Web is stored in RDBs. Therefore, bridging the conceptual gap between the relational model and RDF is needed to make the data available on the Semantic Web. In addition, recent research has shown that Semantic Web technologies are useful beyond the Web, especially if data from different sources has to be exchanged or integrated (*e.g.*, [Patel et al., 2009, Ma et al., 2009, Langegger et al., 2008]).

Many approaches were explored to make relational data available to Semantic Web-enabled applications. Depending on the requirements, these approaches introduced mapping languages that range from simple and pragmatic to highly specific or general-purpose. This lead to the ongoing standardization effort of the World Wide Web Consortium (W3C) carried out in the RDB2RDF Working Group[1] (WG). The mission of the RDB2RDF WG as defined in their charter [Halpin and Herman, 2009] is *to standardize a language for mapping relational data and relational database schemas into RDF and OWL*. Although a standard mapping language is under development, we argue in

[1] `http://www.w3.org/2001/sw/rdb2rdf/`

this paper that alternative languages still have a right to exist as they may provide features or simplicity required in certain use cases that the standard mapping language cannot provide or explicitly excludes.

The goal and contribution of this paper is to provide a feature-based comparison of the state-of-the-art RDB-to-RDF mapping languages. It should act as a guide in selecting an RDB-to-RDF mapping language for a given application scenario and its requirements *w.r.t.* mapping features. Our comparison framework is based on use cases and requirements for mapping RDBs to RDF as identified by the RDB2RDF WG. We apply this comparison framework to the state-of-the-art RDB-to-RDF mapping languages and report the findings in this paper. As a result, our classification proposes four categories of mapping languages: direct mapping, read-only general-purpose mapping, read-write general-purpose mapping, and special-purpose mapping. We further provide recommendations for selecting a mapping language.

The remainder of this paper is structured as follows. Section 2.2 presents existing surveys related to RDB-to-RDF mapping and Section 2.3 introduces the mapping languages covered in this comparison. The framework we use for comparing the mapping languages is defined in Section 2.4. Features are extracted from the document *Use Cases and Requirements for Mapping Relational Databases to RDF* [Prud'hommeaux and Hausenblas, 2010] produced by the RDB2RDF WG. Additional features which recently gained attention in the research community [Hert et al., 2010a, Berners-Lee et al., 2009, Bizer et al., 2011] complement the comparison framework. In Section 2.5, we apply the comparison framework to the presented mapping languages and we discuss the results on a feature-by-feature basis. Section 2.6 classifies the mapping languages into four categories and provides recommendations for selecting a mapping language.

2.2 Related Work

The W3C RDB2RDF Incubator Group[2] (XG) produced *A Survey of Current Approaches for Mapping of Relational Databases to RDF* [Sahoo et al., 2009] as part of their mission. It surveys current techniques, tools, and applications for mapping between RDBs and RDF. A survey reference framework is defined that covers aspects such as mapping creation, representation and accessibility, application domain, support for data integration as well as the implementation of the mapping, *i.e.*, static Extract Transform Load (ETL) versus dynamic query translation. Compared to this paper the survey of the RDB2RDF XG has a different scope. While we focus on the individual *features* of the mapping languages, the RDB2RDF XG survey is focused on the overall approach and implementation of the mappings. The mapping *languages* are only briefly addressed on a higher level and are not compared on a feature-by-feature basis.

The W3C Semantic Web Advanced Development for Europe[3] (SWAD-Europe) dedicates two brief sections of their Deliverable 10.2 [Beckett and Grant, 2003] to RDB-to-RDF mapping. It contains general remarks on RDB-to-RDF mapping and refers to two implemented mapping tools. Due to the early publication of this report in 2003, it could not include many of the mapping languages covered in this paper because they were developed and/or published after the release of the SWAD-Europe deliverable.

2.3 Mapping Languages

In this section, we briefly introduce the mapping languages covered by this comparison. To be included, a mapping language needs to *a)* have a clear focus on mapping RDBs (*i.e.*, other tabular data such as spreadsheets are not covered) and *b)* be general applicable (*i.e.*, not a domain-specific solution).

[2] `http://www.w3.org/2005/Incubator/rdb2rdf/`
[3] `http://www.w3.org/2001/sw/Europe/`

Direct Mapping: In [Berners-Lee, 2009b], a direct approach for mapping RDBs to the Semantic Web is proposed. It maps relational tables to classes in an RDF vocabulary and the attributes of the tables to properties in the vocabulary. The goal is to expose an RDB on the (Semantic) Web to make extra statements about it. The URIs of the instances as well as those of the vocabulary classes are generated automatically based on the RDB schema and data.

The focus of [Hu and Qu, 2007] is on the automatic discovery of mappings and not on their representation. The result is a simple table-to-class and attribute-to-property mapping extended with heuristics to find implicit subclass relationships in the RDB schema.

SquirrelRDF [SquirrelRDF, 2006] is another implementation of the direct mapping as proposed in [Berners-Lee, 2009b]. Its mapping is raw, *i.e.*, the classes and properties of the target RDF vocabulary are generated from the names in the RDB schema. Mapping to a domain ontology is postponed to a later stage and is performed by RDF-based tools.

All three mappings use a similar direct approach for RDB-to-RDF mapping. We therefore summarize them under the term *Direct Mapping*.

eD2R: The case study described in [Barrasa et al., 2003] uses eD2R for the RDB-to-RDF mapping. eD2R is an extension of D2R MAP [Bizer, 2003] with the goal of covering mapping situations involving databases that are lightly structured or not in first normal form [Garcia-Molina et al., 2008]. The mappings are based on SQL queries that extract records from the RDB and transformation functions that can be applied to the extracted values. Existing vocabularies can be reused. eD2R extends the XML-based syntax of D2R MAP to represent the mappings.

R_2O: R_2O [Barrasa et al., 2004] is an extensible and fully declarative language to describe mappings between RDB schemata and ontologies implemented in RDFS or OWL. It is assumed that the RDB and ontology models are preexisting. R_2O is aimed at situations where the similarity

between the ontology and the RDB model is low. It has been conceived to be expressive enough to cope with complex mapping cases where one model is richer, more generic/specific, or better structured than the other. Mappings are expressed in a XML-based syntax.

Relational.OWL: In [de Laborda and Conrad, 2005], a OWL-based representation format for relational data and schema components, called Relational.OWL, is introduced. It defines a OWL Full ontology to describe the schema and data of an RDB. The target application of this mapping is data exchange in peer-to-peer databases.

Virtuoso RDF Views: The Virtuoso Universal Server by Openlink Software features RDF Views [Erling and Mikhailov, 2007, OpenLink Software, 2008] to expose relational data on the Semantic Web. It consists of a declarative Meta Schema Language for defining the mapping of SQL data to preexisting RDF vocabularies. At the most basic level, Virtuoso RDF Views transform the result set of a SQL SELECT query into a set of triples. The Meta Schema Language resembles SQL DDL from a syntax point of view.

D2RQ: D2RQ [Bizer and Seaborne, 2004, Bizer et al., 2009] is a mapping language and platform for treating non-RDF relational databases as virtual RDF graphs. Its aim is to expose RDBs on the Semantic Web to provide access via SPARQL queries and Linked Data. Existing RDF vocabularies can be reused. The mappings are expressed in RDF and formally defined by an RDFS schema. It is the successor to the XML-based D2R MAP [Bizer, 2003].

Triplify: Triplify [Auer et al., 2009] is a light-weight approach to publish Linked Data from RDBs. It is based on mapping HTTP-URI requests onto RDB queries and translating the resulting relations into RDF statements. The main motivation of Triplify is that the majority of information on the Web is already stored in structured form (*i.e.*, as data in RDBs) but published as HTML by Web applications (*e.g.*, CMS, Wiki, Blog). Map-

ping the RDB schemata of such popular Web applications results in a boost of Semantic Web adoption as these Web applications are deployed many times. Triplify mappings are implemented as PHP scripts.

R2RML: R2RML [Das et al., 2010] is the mapping language of the ongoing work by the W3C RDB2RDF WG to standardize RDB-to-RDF mappings. The goal is to define a vendor-independent mapping language for read-only data access.

R3M: R3M [Hert et al., 2010b] is the mapping language of the ONTOACCESS[4] mediation platform [Hert, 2009]. As an update-aware mapping language it enables bidirectional RDF-based access to the RDB, *i.e.*, read and *write* access is supported. R3M employs an RDF-based syntax that contains the mappings of tables to classes and attributes to properties as well as information about integrity constraints.

2.4 Comparison Framework

In this section, we introduce the framework used for comparing the RDB-to-RDF mapping languages presented above. It is based on the document *Use Cases and Requirements for Mapping Relational Databases to RDF* [Prud'hommeaux and Hausenblas, 2010] of the W3C RDB2RDF Working Group[5] (WG). Extensions are made in the area of RDF-based write access, a feature not addressed by the W3C RDB2RDF WG that lately gained attention from the research community [Hert et al., 2010a, Berners-Lee et al., 2009, Bizer et al., 2011].

A direct approach for mapping RDBs to the Semantic Web was proposed in [Berners-Lee, 2009b]. It maps (physical) relational tables to classes in an RDF vocabulary and relational attributes to properties in that vocabulary. We consider this to be the most basic mapping and we require that a mapping language supports at least this kind of mapping to be included in the comparison.

[4] http://ontoaccess.org/
[5] http://www.w3.org/2001/sw/rdb2rdf/

Therefore, these two features are not explicitly represented in the comparison framework but are assumed to hold implicitly.

We now enumerate the features that define the framework our comparison of RDB-to-RDF mapping languages is based on.

F1 Logical Table to Class: A logical table is defined as a SQL view already stored in the RDB system or the result of an ad-hoc SQL query, both resulting in a table that is not necessarily stored physically in the RDB. Feature F1 enables the mapping of such a logical table to a class in the RDF vocabulary.

F2 M:N Relationships: RDBs require a special construct called link (or join) tables to represent M:N relationships among concepts. RDF, however, does not require such helper constructs. Therefore, link tables should be mapped to RDF properties instead of classes. Feature F2 enables the mapping of link tables to properties in the RDF vocabulary.

F3 Project Attributes: Tables in an RDB may contain attributes that should not be part of the RDF representation (*e.g.*, irrelevant or sensitive attributes such as passwords). A mapping should project only the required attributes to the RDF representation. Feature F3 enables projecting a subset of the attributes in the mapping.

F4 Select Conditions: RDB tables may contain records that should not be part of the RDF representation (*e.g.*, outdated data). A mapping should support the definition of a condition that is evaluated for each record to decide about its inclusion in the RDF representation. Feature F4 enables the definition of such select conditions in the mapping.

F5 User-defined Instance URIs: Records in the RDB are converted to RDF instances identified by an URI. These instance URIs can be automatically generated based on the RDB schema and data or the user of the RDB-to-RDF mapping may be able to define the (syntactic) form of the generated URIs. Feature F5 enables user-defined instance URIs.

F6 Literal to URI: URIs as a datatype are typically not supported in an RDB system. Therefore, values representing URIs are stored as character literals (*e.g.*, email addresses). Such literal values should be converted to valid URIs in the RDF representation. Feature F6 enables the generation of URIs from literal values.

F7 Vocabulary Reuse: The vocabulary terms that an RDB schema is mapped to can be generated automatically (based on the names of tables and attributes in the RDB schema) or existing RDF vocabularies can be reused. Feature F7 enables the mapping to existing RDF vocabulary terms.

F8 Transformation Functions: Literal values may require a different (syntactic) representation in RDF (*e.g.*, temperature in Centigrade vs. Fahrenheit). Transformation functions may be defined to provide the conversion of values between the RDB and RDF representations. Feature F8 enables support for such transformation functions.

F9 Datatypes: Datatypes of literal values are an important feature in RDB systems and RDF. Although there exist mappings [ISO/IEC, 2006] of common SQL datatypes to the XML datatypes [Biron and Malhotra, 2004] used in RDF, the information about datatypes might be of value. Feature F9 enables the explicit representation of datatype information in the mapping.

F10 Named Graphs: RDF data sets may consists of multiple named graphs. A mapping may therefore assign certain parts of an RDB to a specific named graph. Feature F10 enables the support of named graphs in the RDB-to-RDF mapping.

F11 Blank Nodes: Blank nodes are used in RDF to represent instances that have no RDF URI reference identifier but are distinct in an RDF graph [Manola and Miller, 2004], *i.e.*, they are a form of existential quantification [Hayes, 2004]. One common usage of blank nodes is in structured property values (*e.g.*, structuring an address consisting of street, postal code, and city). In the case of RDB-to-RDF mapping they may also

be used to represent RDB records without a primary key. Feature F11 enables support for generating blank nodes.

F12 Integrity Constraints: Integrity constraints provide a basic mechanism of semantics in RDBs. We distinguish between key constraints (primary key, foreign key) and other constraints (not null, unique, check). Feature F12 enables the explicit description of constraints in a mapping language.

F13 Static Metadata: Static metadata may be added to the RDF representation that has no direct counterpart in the RDB (*e.g.*, provenance or licensing information). Schema-level triples such as *rdf:type* triples and triples originating from the target RDF vocabulary are, however, not in the scope of this feature. Feature F13 enables the definition of static metadata.

F14 One Table to *n* Classes: Mapping a single table to multiple classes in the RDF vocabulary may be necessary if the RDB schema is *a)* not normalized or *b)* concept specialization is encoded as an attribute of the table. This results in two mapping cases: *a)* the table is mapped multiple times, each time with a subset of the attributes and *b)* records of the table are mapped to a different class depending on the value of a specific discriminator attribute. Feature F14 enables the mapping of one table to n classes.

F15 Write Support: RDF data, including mapped RDBs, are often accessed in a read-only manner (*e.g.*, via SPARQL queries or Linked Data). However, support for write access is required in certain use cases which lately gained attention from the research community [Hert et al., 2010a, Berners-Lee et al., 2009, Bizer et al., 2011]. The requirements of write access should be explicitly addressed in a mapping language. Feature F15 enables explicit support for RDF-based write access to relational data.

Based on this set of fifteen features we compare the mapping languages presented in Section 2.3 and discuss important differences in feature support and the resulting consequences for RDB-to-RDF mapping systems.

2.5 Discussion

In this section, we discuss the RDB-to-RDF mapping languages presented in Section 2.3. The discussion is structured according to the features of the comparison framework introduced in Section 2.4. We mostly limit the discussion to the mapping language that either support a feature partially or not at all. If a mapping language is not mentioned in the discussion of a feature, the reader may assume that it supports the feature. See also Table 2.1 for a summary.

Table 2.1: Summary Table of RDB-to-RDF Mapping Language Comparison

	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15
Direct Mapping	(✔)	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘	✔
eD2R	✔	✔	✔	✔	✔	✔	✔	✔	✔	✘	✔	(✔)	✘	✔	✘
R_2O	✔	✘	✔	✔	✔	✔	✔	✔	✔	✘	✔	(✔)	✘	(✔)	✘
Relational.OWL	(✔)	✘	✔	✘	✘	✘	✘	✘	✔	✘	✔	(✔)	✘	✘	✔
Virtuoso	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	(✔)	✘	✔	✘
D2RQ	✔	✔	✔	✔	✔	✔	✔	✔	✔	✘	✔	(✔)	✔	✔	✘
Triplify	✔	✔	✔	✔	✔	✔	✔	✔	✔	✘	✘	(✔)	✘	✔	✘
R2RML	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	(✔)	✔	✔	✘
R3M	(✔)	✔	✔	(✔)	✔	✔	✔	✔	✔	✘	✘	✔	✘	✔	✔

✔ = full support (✔) = partial support ✘ = no support

F1 Logical Table to Class: Mapping logical tables to ontology classes is supported in one form by all of the mapping languages. This is the case in the mapping of existing views as they can be treated like a physical table. The Direct Mapping does not support mapping the results of an ad hoc query to a class. Relational.OWL does not mention the mapping of logical tables due to its focus on representing the core database schema and data. R3M does not support mapping the results of a query. Mapping of existing views is not prohibited, although this may interfere with R3M's main motivation of bidirectional data access (*cf.* the view update problem [Bancilhon and Spyratos, 1981]).

F2 M:N Relationships: R3M is the only mapping language to provide explicit support for M:N relationships. The Direct Mapping, R_2O, and Relational.OWL do not mention any special support for mapping link tables to ontology properties. The other mapping languages provide implicit support due to their use of SQL (or fragments thereof) as an essential part of the mapping language.

F3 Project Attributes: The Direct Mapping automatically maps every attribute to properties, projecting only a subset of attributes is not intended. Virtuoso, Triplify as well as R2RML delegate the projection of attributes to the SQL queries used for the mapping.

F4 Select Conditions: The Direct Mapping does not provide the selection of rows via a condition. Also Relational.OWL does not allow this. Virtuoso, Triplify, and R2RML again use SQL with its powerful support for conditions to implement this feature. R3M provides limited support for the feature as solely conditions with equality are allowed to preserve the support for bidirectional data access.

F5 User-defined Instance URIs: Instance URIs are generated automatically in the Direct Mapping, hence customization by the user is not supported. Likewise, Relational.OWL with its focus on data exchange does not intend user-defined instance URIs, in fact, blank nodes are used for all instances. Triplify relies on the string concatenation feature of SQL to define custom instance URIs.

F6 Literal to URI: No support for this feature is provided by the Direct Mapping as well as Relational.OWL. All other mapping languages support it either with explicit language constructs or in the case of Triplify and R2RML via the SQL-based mapping.

F7 Vocabulary Reuse: The Direct Mapping itself does not support the reuse of existing vocabulary terms. Properties are generated based on the

attribute names in the database schema. Existing classes can only be used by adding corresponding *rdf:type* statements in an additional step. Relational.OWL does also not provide support for vocabulary reuse.

F8 Transformation Functions: Transformation functions are not in the scope of the Direct Mapping and Relational.OWL. Triplify and R2RML use functions in SQL to transform object values. R3M requires the functions to be defined bidirectionally to retain read and write data access.

F9 Datatypes: The Direct Mapping is the only mapping language that does not describe the explicit representation or the mapping of datatypes.

F10 Named Graphs: Many of the mapping languages (Direct Mapping, eD2R, R_2O, and D2RQ) predate the introduction of named graphs and therefore provide no support for this feature. Others (Relational.OWL, Triplify, and R3M) explicitly choose not to support named graphs for various reasons.

F11 Blank Nodes: Instance URIs in the Direct Mapping are generated automatically for each database record and therefore blank nodes are not supported. In Relational.OWL, all instances are represented as blank nodes because individual URIs are not needed for its main application scenario as a data exchange format. Support for blank nodes is not described in Triplify and R3M.

F12 Integrity Constraints: No support for integrity constraints is available in the Direct Mapping. R3M provides rich support for key and the other constraints. All other mapping languages are limited to key constraints.

F13 Static Metadata: Generating triples on the schema level (*e.g.*, *rdf:type* triples) is supported by many of the mapping languages but is not in the scope of this feature as explained in Section 2.4. Support for static, non-schema-level triples (*e.g.*, provenance or licensing information) is only available in D2RQ and R2RML.

F14 One Table to n Classes: The Direct Mapping as well as Relational.OWL do not allow the mapping of a single table to multiple classes. R_2O provides partial support as mapping a table multiple times with a subset of the attributes is possible while mapping a table multiple times based on the value of a discriminator attribute is not. Virtuoso, D2RQ, Triplify, and R2RML again rely on the power of a SQL-based mapping to support this feature. R3M allows the mapping of one table to n classes in a restricted form. To preserve bidirectional data access, the constraints defined on a table must not be violated in a mapping. For example, if a table contains an attribute with a not null constraint, this attribute must be mapped to each class because it must be set for each record. Otherwise, it is not possible to insert any instances of this class.

F15 Write Support: Support for RDF-based write access to the RDB is influenced by multiple of the other features, but some mapping languages address write support explicitly while others choose a feature set that renders write support impractical. The Direct Mapping could support write access as it represents the RDB schema directly in RDF. However, none of the existing approaches that apply the Direct Mapping consider write access explicitly in their feature set. Relational.OWL with its data exchange background obviously provides write support. R3M as a bidirectional approach does so as well. The other mapping languages were all designed with read-only use cases in mind. Attempts were made to add write support to some of the read-only languages (*e.g.*, D2RQ/Update[6] for D2RQ or [Garrote and Garcia, 2011] for R2RML), but it was also shown that this requires restrictions to the mapping languages resulting in existing mapping definitions to be no longer valid in the restricted approaches.

[6] `http://d2rqupdate.cs.technion.ac.il/`

2.6 Conclusion

In this paper, we presented a feature-based comparison of the state-of-the-art RDB-to-RDF mapping languages.

Based on feature support, the mapping languages are classified into four categories: direct mapping, read-only general-purpose mapping, read-write general-purpose mapping, and special-purpose mapping.

Direct Mapping: The direct mapping, as its name implies, is a direct approach for RDB-to-RDF mapping. Aimed at providing simple means to map RDBs to RDF it does not fully support any of the additional features used in this comparison. Therefore, peculiarities of the relational model such as link tables remain in the RDF representation, although RDF does note require such helper constructs as it provides direct means to model these relationships. The advantage of this mapping is its simplicity to understand and implement the language. It is therefore recommended in application scenarios where a direct representation of the relational schema is acceptable and simplicity is of higher value.

Read-only General-purpose Mapping: Read-only general-purpose mapping languages (Virtuoso, D2RQ, and R2RML) are very similar *w.r.t.* the features they support. The differences are in features not directly related to the expressiveness of the mappings, namely support for named graphs (F10) and static metadata (F13). Virtuoso provides support for named graphs (F10) but not for static metadata (F13). The opposite is the case in D2RQ, no support for named graphs (F10) is provided, but static metadata (F13) can be specified. R2RML supports both features.

These mapping languages enable a highly expressive bridging of the conceptual gap between RDF and the relational model. However, this higher expressiveness also implies an increased complexity that results in an unidirectional mapping, *i.e.*, bidirectional read and write access to the data is impractical. Understanding and implementing the mapping languages may also require a higher learning effort. Due to their high

expressiveness, mapping languages of this category can be recommended for various application scenarios as long as the usage is limited to read-only data access.

In choosing a specific mapping language of this category, aspects not related to feature support may influence the decision such as implementation maturity, standards compliance, or licensing terms.

Read-Write General-purpose Mapping: R3M can be classified as a general-purpose mapping language as well, but it has the additional goal of providing bidirectional (*i.e.*, read and write) data access to the RDB. This explains most of the differences in feature support compared to the read-only general-purpose mapping languages. The most important difference is that in R3M the mapping of logical tables (F1) can not be allowed arbitrarily due to the view update problem [Bancilhon and Spyratos, 1981]. The definition of select conditions (F4) that decide about the inclusion of a record in the RDF representation must also be restricted to preserve write access. Conditions based on inequalities (*e.g.*, less than $<$) result in ambiguities if new data should be inserted. For example, imagine a mapping where a table *person* that represents people of all ages should be (partially) mapped to a concept *Adult* that represents people of the age 18 or older. For this, a select condition must be defined on the attribute *year_of_birth* of the *person* table, namely that its value is less than *1993*. If now a new instance of the concept *Adult* should be inserted into the RDB that does not explicitly contain a value for *year_of_birth*, it remains ambiguous what value should be set for this attribute as any value less than *1993* satisfies the select condition. Therefore, select conditions have to be restricted to equality ($=$) conditions. Furthermore, support for integrity constraints (F12) is extended to other constraints such as not null, unique, and check to enable the detection of invalid write requests, *e.g.*,, detecting missing data for a not null attribute.

In summary, the mapping language of this category provides a more expressive bridging of the conceptual gap between the relational model

and RDF than the mapping approaches of the direct mapping. Peculiarities of the relational model such as link tables are not transferred to the RDF representation. This mapping language is, however, less expressive than the read-only general-purpose mapping languages, but this is required to guarantee support for write access. R3M as the sole mapping language of this category is recommended for application scenarios where RDF-based read and write access to the relational data is needed.

Special-purpose Mapping: Mapping languages such as eD2R, R_2O, and Triplify were developed for specific use cases and are obviously influenced by those use cases in the features they support. This does not necessarily result in a loss of expressiveness or applicability compared to the general-purpose languages. The differences are mostly limited to a few features such as support for named graphs (F10), blank nodes (F11), or static metadata (F13). None of the mapping languages in this category provide support for named graphs (F10) or static metadata (F13). Blank nodes (F11) are supported by eD2R and R_2O but not by Triplify. The use case of Relational.OWL, however, is highly specialized and does therefore neither implement nor require many of the described features.

The mapping languages of this category were developed for their specific application scenarios and are therefore recommended for application in closely related scenarios where the general-purpose mapping languages are not applicable or too complex.

Virtuoso, Triplify, R2RML, and to some extent D2RQ are mapping languages that rely heavily on SQL to implement the mapping. While on the one hand this yields certain advantages, it also entails serious drawbacks. The key benefit is that it allows one to reuse the power of the SQL language in defining views over the relational data. This pushes the main mapping work to the database system and therefore reduces implementation effort. On the other hand, there are two major drawbacks. First of all, in using SQL as the mapping language the semantics of the mapping is hidden in SQL strings

and is therefore not easily accessible, *i.e.*, not without parsing the SQL strings. Second, mappings based on SQL views suffer from the same problem *w.r.t.* write access as standard SQL views, namely the view update problem [Bancilhon and Spyratos, 1981]. History showed that trying to add write access to a mapping or a view definition language is in general impractical (*cf.* discussion of *F15 Write Support* in Section 2.5). Existing mapping or view definition languages would have to be restricted to a subset of the original language to provide general support for write access to the data (*e.g.*, [Bohannon et al., 2006]). This renders existing mapping/view definitions incompatible with the restricted languages. As a result, existing mapping/view definitions need to be rewritten, invalidating one of the top argument for reusing existing mappings. In certain mapping cases it might not even be possible to adapt the mapping because it uses some of the features that render the mapping language read-only.

The situation in the current read-only mapping languages resembles the introduction of SQL views where write access was also not addressed from the beginning. Even the ongoing standardization work by the W3C RDB2RDF WG explicitly defines write access to the data as out of scope [Halpin and Herman, 2009]. This trend leads to a situation where RDB-to-RDF data sets are not on par with native RDF data sets, but limited to read-only application scenarios. Currently, there is no high demand for write access to RDF data as the present SPARQL 1.0 [Prud'hommeaux and Seaborne, 2008] recommendation is limited to read-only queries and no standard data manipulation approach for RDF exists. However, the upcoming SPARQL 1.1 recommendation includes the update language for RDF called SPARQL 1.1 Update [Schenk et al., 2010] and the SPARQL 1.1 Graph Store HTTP Protocol [Ogbuji, 2011], which will increase the demand for write access. Write access should be possible irrespective of the source of the data being a native RDF triple store or a mediated RDB. Approaches such as R3M exist that address this problem.

In summary, we showed that based on feature support the state-of-the-art RDB-to-RDF mapping languages can be classified into four categories: direct mappings that provide simple means to represent RDB schemata and data in RDF; general-purpose mapping languages that provide highly expressive RDB-to-RDF mappings, but are limited to read-only data access; general-purpose mapping languages that are less expressive but enable a bidirectional (*i.e.*, read and write) data access; and special-purpose mapping languages with a feature set tailored to specific application scenarios. We further provided recommendations for selecting a mapping language.

3

Updating Relational Data via SPARQL/Update

Matthias Hert, Gerald Reif, Harald C. Gall
Published at the Workshop on Updates in XML at the
13th International Conference on Extending Database Technology, 2010
DOI: 10.1145/1754239.1754266

Abstract

Relational Databases are used in most current enterprise environments to store and manage data. The semantics of the data is not explicitly encoded in the relational model, but implicitly on the application level. Ontologies and Semantic Web technologies provide explicit semantics that allows data to be shared and reused across application, enterprise, and community boundaries. Converting all relational data to RDF is often not feasible, therefore

we adopt an ontology-based access to relational databases. While existing approaches focus on read-only access, we present our approach OntoAccess that adds ontology-based write access to relational data. OntoAccess consists of the update-aware RDB to RDF mapping language R3M and algorithms for translating SPARQL/Update operations to SQL. This paper presents the mapping language, the translation algorithms, and a prototype implementation of OntoAccess.

3.1 Introduction

Relational Databases (RDBs) are used in most current enterprise environments to store and manage data. While RDBs are well suited to handle large amounts of data, they were not designed to preserve the data semantics. The meaning of the data is implicit on the application level and not explicitly encoded in the relational model. Ontologies and Semantic Web technologies provide explicit semantics in a common framework that allows data to be shared and reused across application, enterprise, and community boundaries [Berners-Lee et al., 2001]. Applying Semantic Web technologies in an enterprise environment enables data processing and exchange on a semantic level. Ontologies and RDF are used to build a semantic layer on top of existing databases that lifts data processing from the syntax to the semantic level. RDF and a shared ontology can be used to exchange data even if the individual relational schemata do not match. The introduction of background knowledge from an ontology can also be valuable in the implementation of a data integration layer on top of multiple relational data sources.

Converting all data in an RDB to RDF is often not feasible due to existing applications that rely on the relational representation of the data. Also, the performance of current triple store implementations remains below RDBs as recent benchmarks show [Bizer and Schultz, 2008a]. Therefore, a mediation approach that performs an on demand translation of Semantic Web requests to SQL is the alternative that preserves the compatibility with existing relational applications while enabling access for ontology-based software to (co-)operate

on the same data. In addition, mediation allows to further exploit the advantages of the well established database technology such as query performance, scalability, transaction support, and security.

Existing approaches for mapping RDBs to RDF focus on exposing the relational data to the Semantic Web. They provide SPARQL endpoints to query the data, but they neither address data updates nor the explicit application in an enterprise environment. Our contribution in this paper is the ontology-based write access to relational data via SPARQL/Update [Seaborne et al., 2008], the upcoming data manipulation language (DML) of the Semantic Web. We present the update-aware RDB to RDF mapping language R3M and algorithms for translating SPARQL/Update to SQL DML.

The remainder of this paper is organized as follows. Section 3.2 presents an overview of related work. The challenges of ontology-based write access to relational data and our approach ONTOACCESS are presented in Section 3.3. In Section 3.4, we introduce our update-aware RDB to RDF mapping language R3M and Section 3.5 specifies the algorithms for translating SPARQL/Update to SQL DML. Our prototype implementation is briefly described in Section 3.6, while Section 3.7 presents a feasibility study as a first evaluation of our approach. Section 3.8 concludes this paper with an outlook on future work.

3.2 Related Work

Relational.OWL [de Laborda and Conrad, 2005] defines an ontology to represent relational schemata and data in RDF. It maps tables and attributes to terms in that ontology and records information about primary/foreign keys as well as the data types of the attributes. This approach exposes the structure and syntax of the relational schema to the RDF representation and prohibits the direct reuse of existing domain vocabulary. RDQuery [de Laborda et al., 2006] adds a SPARQL interface on top of Relational.OWL that provides an on demand translation of SPARQL queries to SQL.

D2R [Bizer and Cyganiak, 2006, Bizer, 2003] is an approach for publishing RDBs on the Semantic Web. It enables the browsing of relational data as RDF

via dereferencable URIs and also provides an endpoint for SPARQL queries. D2Rs main goal is to provide content for the Web of Data, a web of interlinked data sets expressed in RDF (*cf.* the Linked Open Data initiative[1]).

Virtuoso[2] is a commercial database system from OpenLink Software that features RDF Views [Erling and Mikhailov, 2007] over relational data. A declarative meta-schema language is used to map terms of an ontology to concepts in the database schema. This enables the use of SPARQL as an alternative query language for the relational data. RDF Views are limited to read-only queries, updating the base data through these RDF Views is not supported.

Triplify [Auer et al., 2009] is a light-weight approach to expose information from Web applications (*e.g.*, discussion boards, content management systems) in RDF. It uses a set of application-specific SQL queries to extract data from the underlying RDB to generate RDF data from the results. The SQL queries have to be defined manually for each Web application, but the RDF generation is performed automatically according to a fixed process. Reuse of existing ontologies is possible via result column renaming in the SQL queries.

MASTRO-I [Calvanese et al., 2007a] is an ontology-based data integration approach based on global-as-view (GAV) mappings. The individual source schemata are integrated through ontologies and a relational data federation tool. The mappings to the target ontology rely on SQL queries over the federated source schemata and bindings of the query results to terms in an ontology. Hence, the MASTRO-I approach is limited to read-only data access as unrestricted data manipulations would be affected by the relational view update problem.

The World Wide Web Consortium (W3C) has recognized the importance of mapping relational data to the Semantic Web by starting the RDB2RDF incubator group[3] (XG) to investigate the need for standardization. The XG recommends [Malhotra, 2009] that the W3C starts a working group to define a standard RDB to RDF mapping language. However, they will not address the

[1] `http://linkeddata.org`
[2] `http://virtuoso.openlinksw.com`
[3] `http://www.w3.org/2005/Incubator/rdb2rdf/`

requirements for updating the relational data in a first version of the language.

View updates are a well known problem in database research (*e.g.*, [Bancilhon and Spyratos, 1981, Dayal and Bernstein, 1982, Keller, 1985, Langerak, 1990]). Mapping RDBs to RDF can also be seen as defining RDF views over the relational data, therefore these views may be affected by the view update problem. Research in this area has shown that the requirements of updates have to be considered already in the specification of a view definition language (VDL). If a VDL is constructed to allow only the definition of bijective mappings (*i.e.*, updates on the base data as well as the views can unambiguously be propagated to the opposite side), the hardest problems of the relational view update problem can be avoided (*e.g.*, [Bohannon et al., 2006]).

Object-relational mapping (ORM) is an approach to bridge the conceptual gap between object-oriented systems and the relational data model. ORMs such as Hibernate[4] aim at using existing RDB infrastructure to persist data objects in object-oriented applications. This allows to benefit from established database technology while providing an object-oriented abstraction to the relational model. A mapping language is used to define the mappings of classes and attributes in the object-oriented system to tables and attributes in the RDB. The ORM component then generates the RDB schema according to this mapping and also provides means to store and retrieve objects.

3.3 OntoAccess Approach

ONTOACCESS [Hert, 2009] is our approach for ontology-based access to RDBs that provides read and write access to the relational data. It currently consists of the update-aware mapping language R3M that bridges the conceptual gap between an RDB and an ontology as well as an access interface based on SPARQL that supports the upcoming SPARQL/Update language for data manipulations.

Updating relational data through Semantic Web technologies presents new challenges for mapping languages and mediation tools. The conceptual

[4] `http://www.hibernate.org`

gap between the relational model and RDF (tuples vs. triples) causes that constraints from the RDB are transferred to the Semantic Web layer. As a consequence, some update requests are no longer valid compared to their application in a native triple store. The tuple-oriented nature of the relational model requires that a certain amount of data is known about each entity (*i.e.*, attributes declared as mandatory). This and other requirements can be enforced in the database schema with integrity constraints that may not be equally reflected in ontologies and RDF, especially if existing vocabularies are reused. However, to enable ontology-based write access to RDBs these constraints must be respected and errors resulting from constraint violations should be handled appropriately. If information about these constraints is stored in the mapping, it can be used to detect invalid update requests and to provide semantically rich feedback to the client.

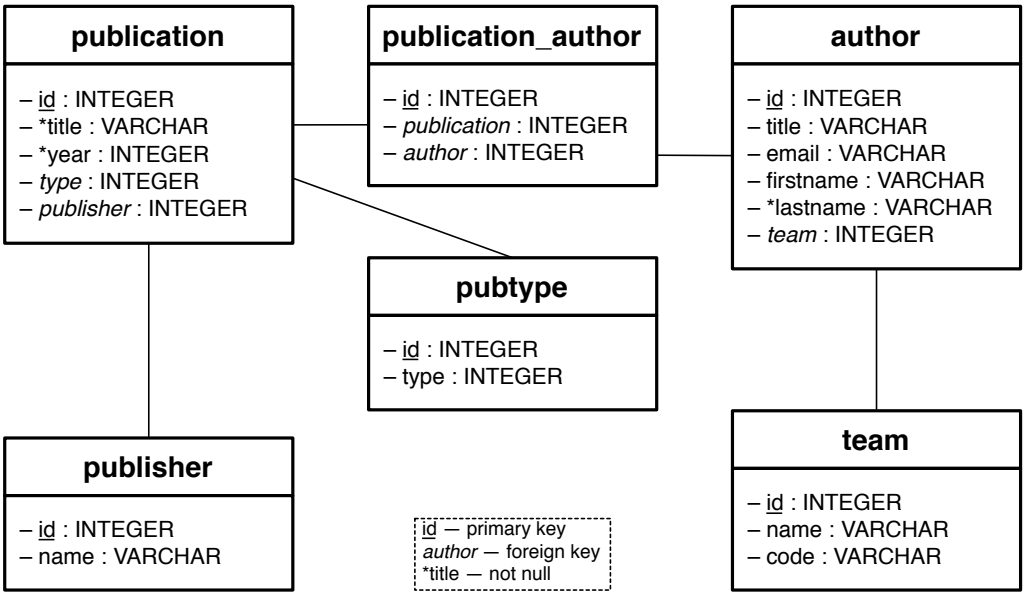

Figure 3.1: RDB Schema of the Publication Use Case

We take the RDB schema of a publication system as the use case for this paper. The database stores information about authors and their publications. Figure 3.1 depicts the database schema used in this example with the tables,

their attributes and data types. Each table has a distinct primary key called *id* of type integer. The *publication* and *author* tables represent the main concepts in the use case. A publication is composed of a *title*, a publication *year*, a publication *type*, and a *publisher*. While *title* and *year* are data attributes, *type* and *publisher* are foreign keys to the tables *pubtype* and *publisher* respectively. Each of those tables contains one textual attribute as a label for the publisher/the type of the publication. All valid publications must have a title and a publication year, therefore the corresponding two attributes have a `NOT NULL` constraint. An author consists of a *title*, an *email* address, a *firstname*, a *lastname*, and an affiliation to a research *team*. A valid author must have at least a last name, therefore a `NOT NULL` constraint is defined on the *lastname* attribute. The *team* attribute is a foreign key to the table of the same name, while the rest of the attributes contain data values. The *team* table stores information about research groups, in particular the *name* of the team and a *code* for abbreviation. The table *publication_author* is a link table that represents the N:M relationship between *publication*s and *author*s.

Figure 3.2 depicts the domain ontology for our example. We reused vocabulary from the Friend of a Friend (FOAF) project[5] and the Dublin Core (DC) metadata standard[6] to form this ontology. We also added our own ontology elements (ONT) if there were no adequate terms in the existing two vocabularies. The figure shows the five classes of our domain ontology as well as the properties that are used with each class and their respective range.

3.4 RDB to RDF Mapping Language

R3M is an update-aware RDB to RDF mapping language that records additional information of the database schema to support data manipulations and to detect invalid update requests during the translation process. Updatability and simplicity were two of the main design goals of this mapping language. It is expressed in RDF and uses the R3M ontology to model the mappings

[5] `http://www.foaf-project.org/`
[6] `http://dublincore.org/`

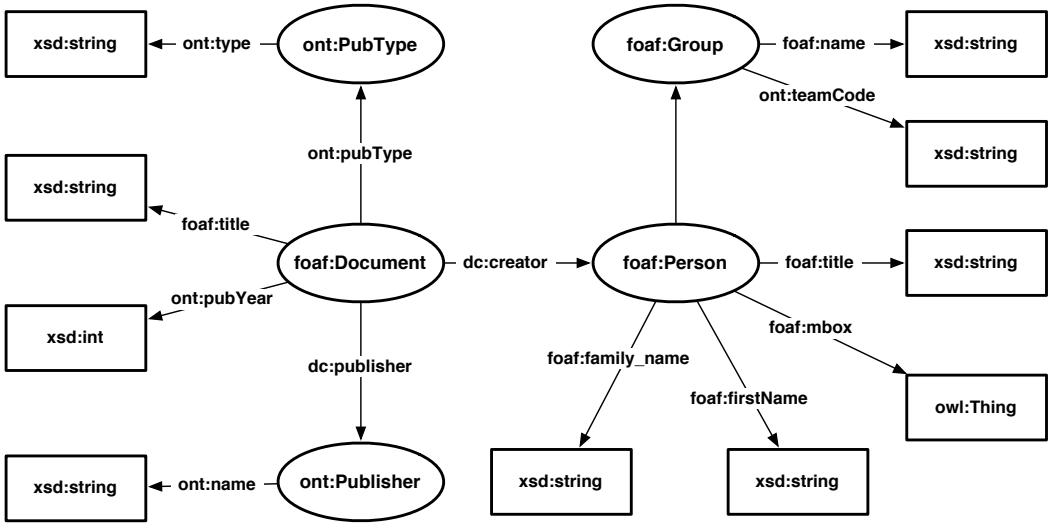

Figure 3.2: Domain Ontology

between terms of a domain ontology and the database schema as well as to record additional information about the schema and its integrity constraints.

The mapping employs the approach where database tables are mapped to ontology classes and attributes to properties. This means, each database table representing a concept in the application domain is mapped to an ontology class representing the same concept. Likewise, each database attribute that constitutes a relationship between an entity and a data value (or another entity) is mapped to an ontology property that links instances of a class to literal values (or other instances). Thereby, each row in a database table is mapped to a set of RDF triples. One triple identifies the entity that is represented by this row as an instance of the class the corresponding table is mapped to. Then, there is in general one triple for each table attribute that relates the instance to a data value or another instance (*e.g.*, foreign keys). Link tables are used in RDBs to describe N:M relationships among relations. In RDF, such auxiliary constructs are not needed, which is why R3M features explicit support to map these tables to object properties instead of classes.

The root element of a mapping in R3M is called *DatabaseMap* (Listing 3.1). It abstractly represents the database and contains access information for the mediator (lines 2 to 5). Optionally, a URI prefix can be specified (line 6) that is used in generating the instance URIs of all the classes defined in the mapping. The URI of an instance is composed of two parts, the mapping-wide URI prefix defined here and an individual URI pattern defined in each *TableMap*. The main purpose of this mapping-wide URI prefix is to ease the definition of mappings similar to the prefix mechanism in XML Namespaces.[7] Finally, all tables that belong to this database schema are listed as *TableMap*s (lines 7 to 8).

A *TableMap* represents the mapping of an individual database table (Listing 3.2). It contains the name of the table (line 2) and the ontology class it is mapped to (line 3). The URI pattern (line 4) is appended to the mapping-wide URI prefix to generate the instance URIs for this class or overrides it if the pattern itself forms a valid URI (*i.e.*, if it starts with `http://`, `mailto:`, etc.). Attribute values from the database table can be included in the pattern by specifying the name of the attribute between double percentage signs. Typically, at least the primary key attributes are included in the URI pattern (*e.g.*, %%*id*%% where *id* is the name of the primary key attribute). A *TableMap* further contains a list of *AttributeMap*s (lines 5 to 8) which map attributes of this table to properties in the ontology.

Each attribute of a database table is represented by an *AttributeMap* (Listings 3.3) that contains the name of the attribute in the database schema (line 2) as well as the name of the ontology property it is mapped to (line 3). Depending on the type or value of the attribute, the property can be an Object- or a DataProperty. This is reflected in the mapping vocabulary as either `r3m:mapsToObjectProperty` or `r3m:mapsToDataProperty`. Additionally, an *AttributeMap* includes information about constraints defined on the attribute (*e.g.*, that it is a foreign key and the table it references; lines 4 and 5). In the current implementation, the following constraints are supported: `r3m:PrimaryKey`, `r3m:ForeignKey`, `r3m:NotNull`, and `r3m:Default`.

[7] `http://www.w3.org/TR/xml-names11/`

```
map:database  a  r3m:DatabaseMap ;
   r3m:jdbcDriver "com.mysql.jdbc.Driver" ;
   r3m:jdbcUrl    "jdbc:mysql://localhost/db" ;
   r3m:username   "user" ;
   r3m:password   "pw" ;
   r3m:uriPrefix  "http://example.org/db/" ;
   r3m:hasTable   map:author, map:publication, map:publisher,
                  map:publication_author, map:team, map:pubtype .
```

Listing 3.1: Example *DatabaseMap*

```
map:author  a  r3m:TableMap ;
   r3m:hasTableName  "author" ;
   r3m:mapsToClass   foaf:Person ;
   r3m:uriPattern    "author%%id%%" ;
   r3m:hasAttribute  map:author_id ,
                     map:author_title, map:author_email,
                     map:author_firstname, map:author_lastname,
                     map:author_team .
```

Listing 3.2: Example *TableMap*

```
map:author_team  a  r3m:AttributeMap ;
   r3m:hasAttributeName      "team" ;
   r3m:mapsToObjectProperty  ont:team ;
   r3m:hasConstraint         [ a r3m:ForeignKey ;
                               r3m:references map:team . ] .
```

Listing 3.3: Example *AttributeMap*

```
map:publication_author  a  r3m:LinkTableMap ;
   r3m:hasTableName          "publication_author" ;
   r3m:mapsToObjectProperty  dc:creator ;
   r3m:hasSubjectAttribute   map:pa_publication ;
   r3m:hasObjectAttribute    map:pa_author .
```

Listing 3.4: Example *LinkTableMap*

A *LinkTableMap* is provided to map link tables to properties in the ontology (Listing 3.4). It specifies the name of the link table in the database (line 2) and the object property it is mapped to (line 3). A link table always contains two foreign key attributes that point to the tables of the N:M relationship. Therefore, a triple with the property representing this link table has a subject and an object mapped from two tables. The attribute pointing to the table of the subject is represented as the subject attribute (line 4) and the attribute pointing to the table of the object as the object attribute (line 5). They link to *AttributeMap*s that are not mapped to any property but record the names of the attributes and the tables they reference (*e.g.*, Listing 3.5).

```
1 map:pa_author  a  r3m:AttributeMap ;
2    r3m:hasAttributeName  "author_id" ;
3    r3m:hasConstraint     [ a r3m:ForeignKey ;
4                             r3m:references map:author . ] .
```

Listing 3.5: Example *AttributeMap* (not mapped)

A basic R3M mapping can be generated automatically from the database schema if it explicitly provides information about foreign key relationships. The only part of the mapping definition that cannot easily be automated is the assignment of domain ontology terms to the individual concepts in the database. However, (graphical) tool support can and will be provided to further decrease the user's effort in defining a mapping.

3.5 SPARQL/Update to SQL DML

SPARQL [Prud'hommeaux and Seaborne, 2008] is the W3C recommendation of a query language for the Semantic Web. It is currently limited to read-only access to RDF data as it does not provide any means to insert, delete, or modify data. The Semantic Web community made efforts to close this gap, which lead to the SPARQL/Update [Seaborne et al., 2008] proposal for an RDF data

manipulation language. SPARQL/Update does also serve as the basis for the update functionality in the relaunched W3C SPARQL working group (WG).[8] The proposed version of SPARQL/Update consists of three update operations: (1) `INSERT DATA` (Listing 3.6) to insert new triples into an RDF graph; (2) `DELETE DATA` (Listing 3.7) to remove known triples from a graph; and (3) `MODIFY` (Listing 3.8) to delete and/or insert data based on triple templates that are matched against a triple pattern in a shared `WHERE` clause. The `MODIFY` operation basically corresponds to two SPARQL `CONSTRUCT` queries (with the same `WHERE` clause) where the resulting RDF triples get removed from and added to the data.

```
1 INSERT DATA {
2     triples
3 }
```

Listing 3.6: `INSERT DATA`

```
1 DELETE DATA {
2     triples
3 }
```

Listing 3.7: `DELETE DATA`

```
1  MODIFY
2  DELETE {
3      template
4  }
5  INSERT {
6      template
7  }
8  WHERE {
9      pattern
10 }
```

Listing 3.8: `MODIFY`

Angles and Gutierrez showed in [Angles and Gutierrez, 2008] that SPARQL has the same expressive power as relational algebra and consequently that SPARQL can be fully translated to SQL. From these findings and the fact that SPARQL/Update is based on SPARQL follows that SPARQL/Update is also fully translatable to SQL DML, albeit not directly as we will see later.

3.5.1 INSERT DATA / DELETE DATA

`INSERT DATA` and `DELETE DATA` operations consist of sets of triples that are either added to or removed from the existing data. Their translation to SQL is therefore very similar and differs mainly in the type of SQL statement

[8] `http://www.w3.org/2009/sparql/wiki/Main_Page`

Algorithm 1 RDF triples to SQL DML translation

```
1: subjectGroups ← groupTriples(triples)
2: for all subjectGroup in subjectGroups do
3:    table ← identifyTable(subjectGroup.getSubject())
4:    if check(subjectGroup, table) is true then
5:       sql ← generateSQL(subjectGroup, table)
6:       statements.add(sql)
7:    else
8:       error()
9:    end if
10: end for
11: sortedSql ← sortSQL(statements)
12: executeSQL(sortedSql)
```

that is generated. It is important for the understanding of the translation algorithm to recall how a database schema is mapped to an ontology: tables representing domain concepts are mapped to classes, while attributes and link tables are represented as ontology properties. We will use the `INSERT DATA` operation depicted in Listing 3.9 as an example to explain the translation algorithm (Algorithm 1). In the first step (line 1), the triples need to be grouped according to equal subjects as these triples all represent data about the same entity and therefore target the same table. The triples in our example operation all use the same subject, hence this step returns one group containing all original triples. Each such group is then handled individually (line 2). In step 2 (line 3), the table affected by this group of triples is identified through the URI of their subject. The subject URI in our example is `http://example.org/db/author1`. If we recall the mapping (*cf.* Listing 3.1 and Listing 3.2), we find that this URI matches the pattern `http://example.org/db/author%%id%%` and therefore identifies the table as *author*. Further, we can extract the value *1* for the primary key attribute *id*. Next, the validity of the request is checked in step three (line 4), *i.e.*, it is tested if the data in the request meets the constraints in the relational schema. For instance, in the case of an `INSERT DATA` operation a triple must be present containing a property for every corresponding database attribute that has a `NotNull` constraint but

no `Default` value. This requirement is trivially meet in our `INSERT DATA` operation as it contains triples with properties matching every attribute of the *author* table. Step four (line 5) generates the respective SQL statement by looking up the properties in the corresponding *TableMap* of the current subject and then adding the attribute name as well as the value extracted from the triple's object to the SQL statement. In the example this means for instance that the property `ont:team` is looked up and matched to the *team* attribute (*cf.* Listing 3.3). The attribute name is added to the SQL statement together with the extracted value from the object, namely *5*. The other triples are processed likewise. Steps 2 to 4 are repeated for each group of triples and the generated SQL statements are collected (line 6). After all groups are processed, in step five (line 11) the collected SQL statements are sorted according to the foreign key relationships among the affected tables. Although, from a theoretical point of view this is not necessary if all statements are executed in the context of a single transaction, existing RDB systems check constraints such as referential integrity already during a transaction. Consequently, executing the generated statements in an arbitrary order may result in the failure of the transaction whereas their execution in the sorted order would succeed. Sorting in our example is trivial as there is only one SQL statement. The sixth and last step (line 12) executes the SQL statements in the previously generated sort order. All generated SQL statements that correspond to a single SPARQL/Update operation are executed within the context of one database transaction to ensure the atomicity of the SPARQL/Update operation. Listing 3.10 shows the translated SQL `INSERT` statement generated from our example SPARQL/Update `INSERT DATA` operation.

`INSERT DATA` The `INSERT DATA` operation of SPARQL/Update can be translated to SQL DML according to the algorithm described in the prior section. Depending on the state of the database, the translation results in either an `INSERT INTO` or an `UPDATE` SQL statement. The triple-oriented nature of RDF permits to insert only the minimal data about an entity with a first `INSERT DATA` operation (*e.g.*, just the last name of an author) and later add

```
1  PREFIX  foaf:  <http://xmlns.com/foaf/0.1/>
2  PREFIX  ont:   <http://example.org/ontology#>
3  PREFIX  ex:    <http://example.org/db/>
4
5  INSERT DATA {
6     ex:author6  foaf:title        "Mr" ;
7                 foaf:firstName    "Matthias" ;
8                 foaf:family_name  "Hert" ;
9                 foaf:mbox         <mailto:hert@ifi.uzh.ch> ;
10                ont:team          ex:team5 .
11 }
```

Listing 3.9: Example `INSERT DATA` Operation

```
1  INSERT INTO author(id, title, firstname, lastname, email, team)
2  VALUES (6, 'Mr', 'Matthias', 'Hert', 'hert@ifi.uzh.ch', 5);
```

Listing 3.10: Translated SQL `INSERT` Statement

```
1  MODIFY
2  DELETE {
3      ?x  foaf:mbox  ?mbox .
4  }
5  INSERT {
6      ?x  foaf:mbox  <mailto:hert@example.com> .
7  }
8  WHERE {
9      ?x  rdf:type          foaf:Person ;
10         foaf:firstName    "Matthias" ;
11         foaf:family_name  "Hert" ;
12         foaf:mbox         ?mbox .
13 }
```

Listing 3.11: Example `MODIFY` Operation

```
1 DELETE DATA {
2     ex:author6  foaf:mbox  <mailto:hert@ifi.uzh.ch> .
3 }
4
5 INSERT DATA {
6     ex:author6  foaf:mbox  <mailto:hert@example.com> .
7 }
```

Listing 3.12: Generated `DELETE DATA` and `INSERT DATA` Operations

more information with a second `INSERT DATA` (*e.g.*, the first name and email address of said author). From the RDB perspective, this results first in a SQL `INSERT` statement that creates a new row in a database table for this entity with `NULL` values for all missing attributes (if this complies with the given constraints). The second `INSERT DATA` operation (with the additional data) translates to an SQL `UPDATE` statement that replaces the `NULL`s with actual values. This means, it has to be checked if the entity already exists in the database as this determines the type of the generated SQL statement.

`DELETE DATA` The SPARQL/Update `DELETE DATA` operation is translated according to Algorithm 1 as well. The translation of this operation can also result in two different types of SQL statements depending on the state of the database and the operation. If the data in the operation represents only a subset of the data in the database, the operation is translated to a SQL `UPDATE` statement that sets all mentioned attributes to `NULL` (if this complies with the given constraints). Only if the data in the request operation equals all remaining (*i.e.*, non-null) data in the database, the resulting SQL statement is a `DELETE` that removes the complete row from the database. Therefore, the tuple for the affected entity must be retrieved and analyzed during the translation.

3.5.2 MODIFY

The MODIFY operation in SPARQL/Update cannot directly be translated to SQL as there is no equivalent statement in the SQL DML. MODIFY is an atomic combination of a delete and an insert that in general is not limited to replacing triples, but can also add/remove arbitrary triples. In contrast, the UPDATE statement in SQL is limited to modifying existing data. However, the reuse of the SPARQL grammar in SPARQL/Update makes a translation in multiple steps possible. Algorithm 2 describes how the MODIFY operation is translated to SQL. We will use the MODIFY operation depicted in Listing 3.11 as an example to explain the algorithm. It replaces any email address of the author *"Matthias Hert"* with a new address (*hert@example.com*). First, the MODIFY operation is separated into its individual parts, the INSERT, DELETE, and WHERE clauses (lines 1 to 3). The WHERE part is used to create a SPARQL SELECT query (line 4) that retrieves the data needed for the DELETE and INSERT templates. It is translated to SQL (line 5) and evaluated on the relational data (line 6). Based on the result bindings of that query, one DELETE DATA (line 8) and one INSERT DATA (line 9) operation are built for each binding (line 7) according to the DELETE and INSERT templates of the original MODIFY operation. In our example, the SELECT query returns just one result binding, namely *ex:author6* for the variable *x* and *mailto:hert@ ifi.uzh.ch* for *mbox*. Therefore, one DELETE DATA and one INSERT DATA operations are built based on that binding as shown in Listing 3.12. These are then translated (lines 10 and 11) and executed (lines 12) according to Algorithm 1 described in the previous sections.

In many cases the MODIFY will actually represent a modification of data or rather a replacement of triples. Then, one optimization is possible by omitting those DELETE DATA operations that have a corresponding INSERT DATA, *i.e.*, the triples differ only in their object. In these cases, the delete would set an attribute value to NULL and the insert sets the same attribute to a new value, therefore the delete is redundant and can be omitted.

Algorithm 2 `MODIFY` to SQL DML translation

1: $delete \leftarrow extractDelete(modify)$
2: $insert \leftarrow extractInsert(modify)$
3: $where \leftarrow extractWhere(modify)$
4: $select \leftarrow createSelect(where)$
5: $selectSQL \leftarrow translateSelect(select)$
6: $results \leftarrow executeSQL(selectSQL)$
7: **for all** $binding$ in $results$ **do**
8: $deleteData \leftarrow createDeleteData(delete, binding)$
9: $insertData \leftarrow createInsertData(insert, binding)$
10: $deleteSQL \leftarrow translateDelete(deleteData)$
11: $insertSQL \leftarrow translateInsert(insertData)$
12: $executeSQL(deleteSQL, insertSQL)$
13: **end for**

3.6 Prototype Implementation

Based on our mapping language R3M and the SPARQL/Update to SQL DML translation algorithms described in the previous sections, we developed a prototype that mediates between SPARQL/Update requests and an RDB. Implemented as a HTTP endpoint, it allows clients to remotely manipulate the relational data. Incoming SPARQL/Update operations are parsed from the HTTP requests and forwarded to the translation module. There, the algorithm of Section 3.5.1 is used to generate equivalent SQL statements based on a R3M mapping definition. The translated operation is executed by the database engine and a confirmation or error message is returned to the translation module. This message is then converted to an RDF representation and sent back to the client.

Currently, the implementation is limited to `INSERT DATA` and `DELETE DATA` operations, but support for `MODIFY` and SPARQL queries are under development. Also, a more powerful feedback protocol is planned that will provide semantically rich error information to the client. A future version of the prototype implementing these features will be released to the public.

3.7 Feasibility Study

For a first evaluation of our approach we present a feasibility study based on the RDF schema and the domain ontology introduced in Section 3.3. Table 3.1 summarizes the mapping from tables and attributes of the database schema to classes and properties of the domain ontology. The first column specifies the table and the corresponding class. For each table, column two lists the attributes and the properties they are mapped to.

Table 3.1: Use Case Mapping Overview

table	→	**class**	**attribute**	→	**property**
publication	→	`foaf:Document`	title	→	`dc:title`
			year	→	`ont:pubYear`
			type	→	`ont:pubType`
			publisher	→	`dc:publisher`
publisher	→	`ont:Publisher`	name	→	`ont:name`
pubtype	→	`ont:PubType`	type	→	`ont:type`
author	→	`foaf:Person`	title	→	`foaf:title`
			email	→	`foaf:mbox`
			firstname	→	`foaf:firstName`
			lastname	→	`foaf:family_name`
			team	→	`ont:team`
team	→	`foaf:Group`	name	→	`foaf:name`
			code	→	`ont:teamCode`
publication_author	→	–	–	→	`dc:creator`

The *publication* table is mapped to `foaf:Document`. The attributes *title* and *publisher* are mapped to corresponding properties from DC, while *year* and *type* use properties from our own ontology ONT. The tables *publisher* and *pubtype* as well as their attributes are all mapped to terms of our application-specific ontology. The *author* table is represented as `foaf:Person`. Its attributes are mapped to equivalent concepts from the FOAF vocabulary with the exception of *team* that uses a property from ONT. The table *team* is represented as the

```
1 PREFIX  foaf:  <http://xmlns.com/foaf/0.1/>
2 PREFIX  ont:   <http://example.org/ontology#>
3 PREFIX  ex:    <http://example.org/db/>
4
5 INSERT DATA {
6     ex:team4  foaf:name     "Database Technology" ;
7               ont:teamCode  "DBTG" .
8 }
```

Listing 3.13: Example `INSERT DATA` Operation

```
1 INSERT INTO team (id, name, code)
2 VALUES (4, 'Database Technology', 'DBTG');
```

Listing 3.14: Translated SQL `INSERT` Statement

class `foaf:Group` with its *name* attribute mapped to `foaf:name` and *code* to `ont:teamCode`. The *publication_author* table is a link table that represents the N:M relationship between *publication*s and *author*s. Therefore, as described in Section 3.4, it is not mapped to a class but to the property `dc:creator` instead.

This mapping definition enables our mediation prototype to process SPARQL/ Update operations. In the remainder of this section, we present example SPARQL/Update operations and the translated SQL statements as generated by our prototype.

Listing 3.13 shows a simple SPARQL/Update `INSERT DATA` request that inserts data about a team. It affects only a single database table and is therefore translated to one SQL `INSERT` statement (Listing 3.14).

Listing 3.15 depicts a more complex `INSERT DATA` request. It contains a complete data set, *i.e.*, the request inserts new data into every database table and will therefore generate multiple SQL statements. The order of the triples in the request is irrelevant as the translated SQL statements are sorted based on the foreign key dependencies between the affected tables. Listing 3.16 shows the generated SQL statements sorted in their order of execution.

```
1  PREFIX  foaf:  <http://xmlns.com/foaf/0.1/>
2  PREFIX  dc:    <http://purl.org/dc/elements/1.1/>
3  PREFIX  ont:   <http://example.org/ontology#>
4  PREFIX  ex:    <http://example.org/db/>
5
6  INSERT DATA {
7      ex:pub12  dc:title      "Relational ..." ;
8                ont:pubYear   "2009" ;
9                ont:pubType   ex:pubtype4 ;
10               dc:publisher  ex:publisher3 ;
11               dc:creator    ex:author6 .
12
13     ex:author6  foaf:title        "Mr" ;
14                 foaf:firstName    "Matthias" ;
15                 foaf:family_name  "Hert" ;
16                 foaf:mbox         <mailto:hert@ifi.uzh.ch> ;
17                 ont:team          ex:team5 .
18
19     ex:team5  foaf:name     "Software Engineering" ;
20               ont:teamCode  "SEAL" .
21
22     ex:pubtype4  ont:type  "inproceedings" .
23
24     ex:publisher3  ont:name  "Springer" .
25 }
```

Listing 3.15: Example `INSERT DATA` Operation

Listing 3.17 shows an example SPARQL/Update `DELETE DATA` operation that removes the email address of an existing author. As the respective entry in the *author* table contains more information than just the email address, this request is translated to a SQL `UPDATE` statement (Listing 3.18) according to Algorithm 1 described in Section 3.5.1.

The SPARQL/Update requests presented in this section are just examples, a user is free to phrase arbitrary requests. They will be translated to SQL DML successfully as long as they adhere to the ontology terms from the mapping and respect the constraints of the database schema.

```
1  INSERT INTO team (id, name, code)
2  VALUES (5, 'Software Engineering', 'SEAL');
3
4  INSERT INTO pubtype (id, type)
5  VALUES (4, 'inproceedings');
6
7  INSERT INTO publisher (id, name)
8  VALUES (3, 'Springer');
9
10 INSERT INTO publication (id, title, year, type, publisher)
11 VALUES (12, 'Relational ...', 2009, 4, 3);
12
13 INSERT INTO author(id, title, firstname, lastname, email, team)
14 VALUES (6, 'Mr', 'Matthias', 'Hert', 'hert@ifi.uzh.ch', 5);
15
16 INSERT INTO publication_author (publication, author)
17 VALUES (12, 6);
```

Listing 3.16: Translated SQL `INSERT` Statements

```
1  PREFIX  foaf:  <http://xmlns.com/foaf/0.1/>
2  PREFIX  ont:   <http://example.org/ontology#>
3  PREFIX  ex:    <http://example.org/db/>
4
5  DELETE DATA {
6      ex:author6  foaf:mbox  <mailto:hert@ifi.uzh.ch> .
7  }
```

Listing 3.17: Example `DELETE DATA` Operation

```
1  UPDATE author
2  SET email = NULL
3  WHERE id = 6 AND email = 'hert@ifi.uzh.ch';
```

Listing 3.18: Translated SQL `UPDATE` Statement

3.8 Conclusion and Future Work

In this paper, we presented our approach OntoAccess that enables the manipulation of relational data via SPARQL/Update. We introduced the update-aware RDB to RDF mapping language R3M that captures additional information about the database schema, in particular about integrity constraints. This information enables the detection of update requests that are invalid from the RDB perspective. Such requests cannot be executed by the database engine as they would violate integrity constraints of the database schema. The information can also be exploited to provide semantically rich feedback to the client. Therefore, the causes for the rejection of a request and possible directions for improvement can be reported in an appropriate format.

Future work is planned for various aspects of OntoAccess. Further research needs to be done on bridging the conceptual gap between RDBs and the Semantic Web. Ontology-based write access to the relational data creates completely new challenges on this topic with respect to read-only approaches. The presence of schema constraints in the database can lead to the rejection of update requests that would otherwise be accepted by a native triple store. A feedback protocol that provides semantically rich information about the cause of a rejection and possible directions for improvement plays a major role in bridging the gap. Other database constraints such as assertions have to be evaluated as well to see if they can reasonably be supported in the mapping. Also, a more formal definition of the mapping language will be provided. Furthermore, we will extend our prototype implementation to support the SPARQL/Update `MODIFY` operation, SPARQL queries, and the just mentioned feedback protocol.

4

OntoAccess – An Extensible Platform for RDF-based Read and Write Access to Relational Databases

Matthias Hert, Gerald Reif, Harald C. Gall
Submitted for Publication to the Journal of Web Semantics: Science, Services and Agents on the World Wide Web, 2012

Abstract

Relational Databases (RDBs) are used in most current enterprise environments to store and manage data. The semantics of the data is not explicitly encoded in the relational model, but implicitly at the application level. Ontologies and Semantic Web technologies provide explicit semantics that allows data to be shared and reused across application, enterprise, and community boundaries.

Converting all relational data to RDF is often not feasible, therefore we adopt a mediation approach for RDF-based access to RDBs. Existing RDB-to-RDF mapping approaches focus on read-only access via SPARQL or Linked Data but other data access interfaces exist, including approaches for updating RDF data (*e.g.*, Semantic Web frameworks such as Jena, Sesame, and RDF2Go; ChangeSet). In this paper we present ONTOACCESS, an extensible platform for RDF-based read and write access to existing relational data. It encapsulates the translation logic in the core layer that provides the foundation of an extensible set of data access interfaces in the interface layer. We further present the formal definition of our RDB-to-RDF mapping, the architecture and implementation of our mediator platform, a semantic feedback protocol to bridge the conceptual gap between the relational model and RDF as well as a performance evaluation of the prototype implementation.

4.1 Motivation

Relational Databases (RDBs) are used in most current enterprise environments to store and manage data. While RDBs are well suited to handle large amounts of data, they were not designed to preserve the data semantics. The meaning of the data is implicit at the application level but not explicitly encoded in the relational model.

The Semantic Web provides a common framework that allows data to be shared and reused across application, enterprise, and community boundaries [World Wide Web Consortium, 2011]. Although developed for the Web, these Semantic Web technologies have proven to be useful in other domains as well, especially if data from different sources has to be exchanged or integrated (*e.g.*, [Patel et al., 2009, Ma et al., 2009, Langegger et al., 2008]).

Ontologies and RDF are used to build a semantic layer that lifts data processing and exchange from the syntactic to the semantic level. In existing systems, however, it is not always possible or desirable to convert all relational data to RDF as other applications rely on the *relational* representation. Adapting or replacing these applications would require a prohibitive migration effort.

Therefore, we suggest a mediation approach that performs an on demand translation of Semantic Web requests. This results in a cooperative use of the data in RDF-based as well as relational applications. In addition, mediation allows one to further exploit the advantages of the well established database technology such as query performance, scalability, transaction support, and security.

In the area of Semantic Web technologies, SPARQL [Prud'hommeaux and Seaborne, 2008] is the standard language for querying RDF data but other popular access interfaces exist such as Semantic Web frameworks (*e.g.*, Jena,[1] Sesame[2], RDF2Go[3]) and Linked Data.[4] Further, the Semantic Web currently lacks a standard data manipulation language (DML). SPARQL/Update [Seaborne et al., 2008] was proposed to the World Wide Web Consortium (W3C) as a DML and is being incorporated in the upcoming SPARQL 1.1[5] recommendation. In the meantime, several other approaches for updating RDF data have emerged (*e.g.*, ChangeSet,[6] GUO[7]). Although not approved as standards, these approaches are implemented and used in applications. The upcoming SPARQL 1.1 will further introduce a new data access interface in the form of the *Graph Store HTTP Protocol* [Ogbuji, 2011]. Therefore, an RDB-to-RDF mediator should not be limited to a single data access interface (*e.g.*, SPARQL). Instead, it should be flexible and extensible to support multiple and also future data access interfaces.

The conceptual gap between the relational model and RDF affects the processing of Semantic Web requests if ontology terms are referenced that cannot be mapped to the RDB schema or if a (write) request would violate constraints of the RDB schema. While a read-only query will simply return no results, a write request may not be processable and result in an error. Rejecting such requests may be confusing to an RDF-based client if the request is valid

[1] http://openjena.org/
[2] http://openrdf.org/
[3] http://rdf2go.semweb4j.org/
[4] http://linkeddata.org/
[5] http://www.w3.org/2009/sparql/wiki/Main_Page
[6] http://n2.talis.com/wiki/ChangeSets
[7] http://webr3.org/specs/guo/

in RDF. An RDB-to-RDF mediator should provide feedback about rejected requests in a semantic format understandable by the client, *i.e.*, in RDF.

The contributions of this paper are the extensible RDB-to-RDF mediation platform ONTOACCESS [Hert, 2009] and a formal definition of its RDB-to-RDF mapping. The formal definition includes proofs that mappings expressed in this language are bidirectional, *i.e.*, support for read and write access to the data is provided. We present the architecture and implementation of the ONTOACCESS platform, including a semantic feedback protocol that provides recommendations to the client on how to change invalid write requests for the better.

The remainder of this paper is structured as follows. Section 4.2 presents an overview of related work in the area of RDB-to-RDF mapping. In Section 4.3, we formally define our RDB-to-RDF mapping and present examples in the RDF-based syntax called R3M. Section 4.4 explains the architecture and implementation of our mediator platform in detail. Section 4.5 introduces our semantic feedback protocol that bridges the conceptual gap between the relational model and RDF. The evaluation in Section 4.6 demonstrates the extensibility of our platform and shows that our approach performs comparable or better than the state-of-the-art. It further summarizes a case study we performed in the domain of software evolution analysis. Section 4.7 concludes this paper with a summary and an outlook on future work.

4.2 Related Work

Mapping RDBs to RDF is an active field of research resulting in many mapping languages and approaches (*e.g.*, [Bizer and Seaborne, 2004, Erling and Mikhailov, 2007, Barrasa et al., 2004, Auer et al., 2009, Bizer and Cyganiak, 2006, Das et al., 2010]).

D2R [Bizer and Cyganiak, 2006] is an approach for publishing existing relational databases on the Semantic Web. Based on mappings expressed in the D2RQ [Bizer and Seaborne, 2004] mapping language, it enables browsing the relational data as RDF via dereferenceable URIs (*i.e.*, as Linked Data) and

querying it via SPARQL. Further, D2R provides extensions for the Semantic Web frameworks Jena and Sesame that enable accessing the mapped RDBs via those APIs. However, D2R is limited to read-only data access, updating RDF data is not supported. *D2R/Update*[8] was an attempt to add write access to the D2R approach, but it turned out to be impractical without restricting the existing D2RQ mapping language.

The *Virtuoso* Universal Server features RDF Views [Erling and Mikhailov, 2007] to expose relational data on the Semantic Web. A declarative Meta Schema Language is used for defining the mapping of SQL data to RDF vocabularies. This enables the use of SPARQL as an alternative query language for the relational data. Likewise, Virtuoso implements a Linked Data interface to these views. RDF Views are limited to read-only queries, updating the relational base data is not supported.

R_2O [Barrasa et al., 2004] is an extensible and fully declarative language to describe mappings between relational database schemata and ontologies. R_2O is aimed at situations where the similarity between the ontology and the database model is low. It has been conceived to be expressive enough to cope with complex mapping cases where one model is richer, more generic/specific, or better structured than the other. This high expressiveness renders R_2O mappings read-only.

The W3C has recognized the importance of mapping relational data to the Semantic Web by starting the RDB2RDF Incubator Group[9] (XG) to investigate the need for standardization. The XG recommended [Malhotra, 2009] that the W3C initiates a working group (WG) to define a vendor-independent RDB-to-RDF mapping language. The RDB2RDF WG[10] started its work on R2RML [Das et al., 2010] in late 2009. According to their charter [Halpin and Herman, 2009], the requirements for updating relational data are out of scope and are therefore not addressed by the WG. It was further shown in [Garrote and Garcia, 2011] that adding write support to the R2RML approach is impractical.

[8] `http://d2rqupdate.cs.technion.ac.il/`
[9] `http://www.w3.org/2005/Incubator/rdb2rdf/`
[10] `http://www.w3.org/2001/sw/rdb2rdf/`

pushback[11] is a W3C community project to develop a methodology for writing back changes in RDF data generated by wrappers of Web 2.0 APIs. Write support is enabled with RDF-annotated HTML forms, so-called RDForms, and mappings to the native write interfaces of the Web 2.0 APIs. Since *pushback* is focused on Web applications, direct support for modifying RDBs is not in the scope of the project.

We further refer the reader to the survey [Sahoo et al., 2009] conducted by the W3C RDB2RDF Incubator Group[12] for a detailed overview of existing RDB-to-RDF mapping approaches and to [Hert et al., 2011c] for a feature-based comparison of the mapping *languages*.

4.3 OntoAccess Mapping

Mediation requires a mapping from concepts in an RDB schema to vocabulary terms defined in an ontology. Several of such mapping languages exist. However, thorough investigations revealed that they are unsuitable for RDF-based write access to relational data (*cf.* Section 4.2). Existing mapping languages would need to be extended to include additional information about the database schema required to support write access. For instance, detailed information about foreign key relationships and other integrity constraints is needed to detect invalid write requests. The mapping languages would also need to be restricted to avoid the view update problem [Bancilhon and Spyratos, 1981]. Most existing approaches employ SQL views on the relational schema to define mappings. While this results in a high expressivity, it also means that in the case of write access such mappings are affected by the view update problem, *i.e.*, write access in general is impractical. Therefore, we designed our mapping language R3M to explicitly address write support. It extends the mapping approach described in [Berners-Lee, 2009b] (*cf.* [Hert et al., 2011c] for a feature-based comparison of R3M and other state-of-the-art

[11] `http://esw.w3.org/PushBackDataToLegacySources`
[12] `http://www.w3.org/2005/Incubator/rdb2rdf/`

RDB-to-RDF mapping languages). R3M is sufficient to cover use cases as described in [Fürber, 2009] and, in general, to map normalized RDB schemata (*e.g.*, schemata generated by object-relational mappers such as Hibernate[13]).

In [Hert et al., 2010b], we presented an example-driven definition of our RDB-to-RDF mapping approach. We now introduce a formal definition of our RDB-to-RDF mapping language R3M followed by examples in the RDF-based syntax. The rationale of the formal definition and the proofs is to show that our mapping language R3M is bidirectional and therefore not affected by the view update problem.

Definition 1. *In* ONTOACCESS*, a mapping is defined as an eight-tuple* $\mathcal{M} = \{R, A, L, C, P, \mathcal{T}, \mathcal{A}, \mathcal{L}\}$ *with:*

- $R = \{R_1, ..., R_l\}$ *the set of relations of the source database schema*
- $A = \{A_1, ..., A_l\}$ *the set of attribute sets with* $A_i = \{a_{i1}, ..., a_{im}\}$ *the set of attributes of relation* R_i
- $L = \{L_1, ..., L_k\} \subset R$ *the set of relations that represent N:M relationships with all* L_i *satisfying the condition:* $(\forall a_{ij} \in A_i : a_{ij} \in PK(L_i) \vee a_{ij} \in FK(L_i)) \wedge |FK(L_i)| = 2$ *with* $PK(X)$ *and* $FK(X)$ *being the sets of primary and foreign keys of relation X and* $|FK(X)|$ *being the number of foreign keys of relation X*
- *C the set of classes of the target ontology*
- *P the set of properties of the target ontology*
- $\mathcal{T} : R \backslash L \rightarrow C$ *an injective and surjective partial function mapping tables (without link tables) to ontology classes*
- $\mathcal{A} = \{\mathcal{A}_1, ..., \mathcal{A}_l\}$ *the set of injective and surjective partial functions* $\mathcal{A}_i : A_i \rightarrow P$ *mapping attributes of relation* $R_i \in R \backslash L$ *to ontology properties*

[13] `http://hibernate.org/`

- $\mathcal{L} : L \rightarrow P$ *an injective and surjective partial function mapping link tables to ontology properties*

Lemma 1. *Mappings as defined in Definition 1 are bidirectional.*

Proof. The mapping functions $\mathcal{T}, \mathcal{A}_1, ..., \mathcal{A}_l, \mathcal{L}$ are defined as injective and surjective partial functions, hence there exist unique inverses $\mathcal{T}^{-1}, \mathcal{A}_1^{-1}, ..., \mathcal{A}_l^{-1}$, $\mathcal{L}^{-1}$ that map ontology terms to elements of the database schema. □

The functions $\mathcal{T}$, $\mathcal{A}_1$, ..., $\mathcal{A}_l$, and $\mathcal{L}$ are defined as partial because we do not require that all tables or attributes of a database schema are mapped to terms in the ontology. But defining these functions incautiously may result in invalid mappings. For example, if a foreign key attribute of a table is mapped but the referenced table is not. This results in an invalid mapping since the mapped attribute would be dangling, referencing resources that are not mapped to a table in the database schema. We therefore define the notion of a valid mapping.

Definition 2. *A mapping $\mathcal{M}$ is called **valid** if and only if it is defined according to Definition 1 and if and only if it satisfies the following two conditions:*

(i) $\forall a_{ij} \in A_i : a_{ij} \in NN(R_i) \Rightarrow a_{ij} \in dom\mathcal{A}_i$ *with* $NN(X)$ *being the set of attributes with a* not null *constraint of relation X and* $dom\mathcal{F}$ *the domain of function* $\mathcal{F}$*,* i.e.*, all attributes of a mapped relation with a* not null *constraint must be mapped*

(ii) $\forall a_{ij} \in dom\mathcal{A}_i : a_{ij} \in FK(R_i) \Rightarrow ref(a_{ij}) \in dom\mathcal{T}$ *with* $FK(X)$ *being the set of foreign keys of a relation X,* $ref(Y) : \bigcup A_i \rightarrow R$ *being a function that returns the referenced relation for a foreign key attribute Y, and* $dom\mathcal{F}$ *the domain of function* $\mathcal{F}$*,* i.e.*, if a foreign key attribute is mapped, the relation that it references must also be mapped*

We further define the notion of a complete mapping.

Definition 3. *A mapping $\mathcal{M}$ is called **complete** if and only if the mapping functions $\mathcal{T}$, $\mathcal{A}_1$, ..., $\mathcal{A}_l$, $\mathcal{L}$ are total,* i.e., *all relations and attributes of a source database schema are mapped to classes and properties in the target ontology.*

Lemma 2. *Complete mappings are valid.*

Proof. A complete mapping $\mathcal{M}$ has the properties:

(i) $\forall R_i \in R \backslash L : R_i \in dom\mathcal{T}$, *i.e.*, all relations (without link tables) are mapped to ontology classes

(ii) $\forall A_i \in A : \forall a_{ij} \in A_i : a_{ij} \in dom\mathcal{A}_i$, *i.e.*, all attributes are mapped to ontology properties

(iii) $\forall L_i \in L : L_i \in dom\mathcal{L}$, *i.e.*, all link tables are mapped to ontology properties

and therefore trivially satisfies conditions (i) and (ii) of Definition 2. □

For the remainder of the paper, we assume that mappings are always at least valid if not complete.

We now present selected examples in the RDF-based syntax of our mapping language R3M to further illustrate our mapping approach. The namespace prefixes used in the examples are defined as follows: `r3m` represents our mapping language ontology `http://ontoaccess.org/r3m#` while `ex` is used for the namespace `http://example.com/mapping/` of our example mapping. `foaf` and `dc` represent the namespaces of the well-known *Friend of a Friend*[14] and *Dublin Core*[15] projects. Listing 4.1a) depicts a *TableMap* representing the mapping of a database table to a class in the ontology. The set of all *TableMap*s in a mapping definition implements the mapping function $\mathcal{T}$ of Definition 1. A *TableMap* contains the name of the table (line 2) and the ontology class it is mapped to (line 3). The URI pattern (line 4, abbreviated) is used to generate

[14] `http://xmlns.com/foaf/0.1/`
[15] `http://purl.org/dc/elements/1.1/`

the URIs for instances of this table based on values of table attributes that are specified between double percentage signs (*e.g.*, %%*id*%% where *id* is the name of the primary key attribute). A *TableMap* further contains a list of *AttributeMap*s (lines 5 to 8).

Listing 4.1b) presents an example of an *AttributeMap* that maps a database attribute to a property in the ontology. The set of all *AttributeMap*s in a mapping definition implements the set of mapping functions $\mathcal{A}$ of Definition 1. An *AttributeMap* contains the name of the attribute in the database schema (line 11) and the ontology property it is mapped to (line 12). Additionally, an *AttributeMap* includes information about constraints defined on that attribute (*e.g.*, a *not null* constraint; line 13). Currently the constraints `r3m:PrimaryKey`, `r3m:ForeignKey`, `r3m:NotNull`, and `r3m:Default` are supported.

Listing 4.1c) shows a *LinkTableMap* representing the mapping of a link table to an ontology property. The set of all *LinkTableMap*s in a mapping definition implements the mapping function $\mathcal{L}$ of Definition 1. A *LinkTableMap* specifies the name of the link table in the database (line 16) and the property it is mapped to (line 17). A link table always contains two foreign key attributes that point to the tables of the N:M relationship. These attributes are represented as *AttributeMap*s (line 18 and 19; the definitions of those *AttributeMaps* are not shown in the example) that provide the names of the attributes, the foreign key references to the tables, and the direction of the relationship (from subject to object).

4.4 OntoAccess Platform Architecture and Implementation

The goal of ONTOACCESS is to provide a platform for RDF-based read and write access to data stored in existing RDBs. It supports a broad number of data access interfaces and is extensible for future development in data access approaches. The main idea of the ONTOACCESS platform is to encapsulate the RDB-to-RDF translation logic into basic core operations and thus avoid

```
a) ex:author  a  r3m:TableMap;
     r3m:hasTableName  "author";
     r3m:mapsToClass   foaf:Person;
     r3m:uriPattern    "http://.../author%%id%%";
     r3m:hasAttribute  ex:author_id,
                       ex:author_email,
                       ex:author_firstname,
                       ex:author_lastname.

b) ex:author_email  a  r3m:AttributeMap;
     r3m:hasAttributeName      "email" ;
     r3m:mapsToObjectProperty  foaf:mbox;
     r3m:hasConstraint         [ a r3m:NotNull ].

c) ex:publication_author  a  r3m:LinkTableMap;
     r3m:hasTableName          "publication_author";
     r3m:mapsToObjectProperty  dc:creator;
     r3m:hasSubjectAttribute   ex:pa_publication;
     r3m:hasObjectAttribute    ex:pa_author.
```

Listing 4.1: Example Mappings

repeated implementation of the translation functionality. This simplifies the development of additional data access interfaces and increases the flexibility of the platform.

On the basis of the well-known *CRUD*[16] operations, we define three basic operations for Semantic Web data access: (1) querying for a single triple pattern; (2) adding a set of triples to the data; and (3) removing a set of triples from the data. In principle, it is possible to implement any data access based on these core operations, ignoring for the moment any concerns about performance and atomicity of requests.

Figure 4.1 depicts the architecture of the ONTOACCESS platform. It is split into two layers. The lower part, called core layer, is responsible for the actual RDB-to-RDF translation as well as the interaction with the database system. The upper part, called interface layer, exposes the functionality of the ONTO-

[16] **C**reate, **R**ead, **U**pdate, **D**elete

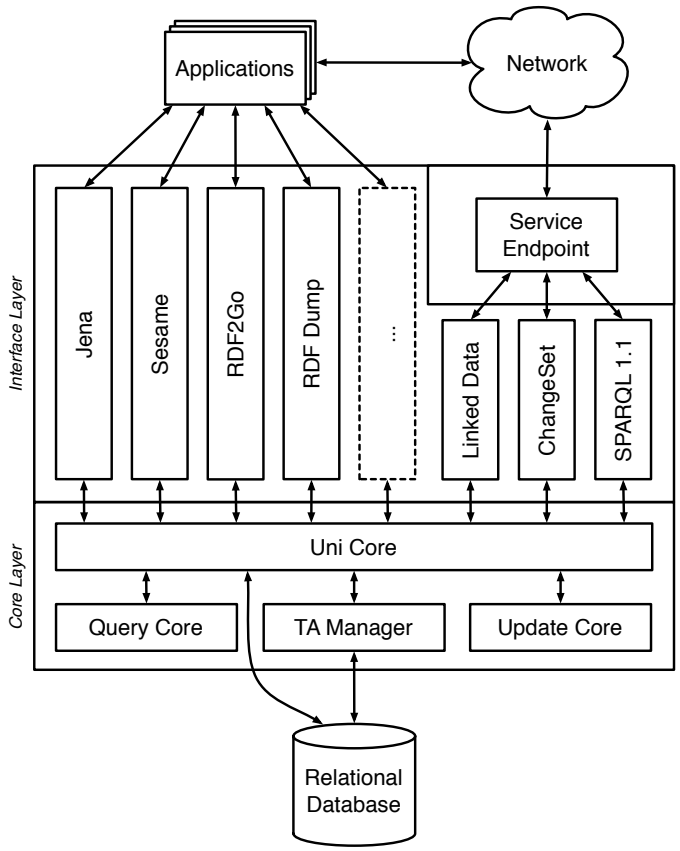

Figure 4.1: Platform Architecture

ACCESS core to the individual data access approaches. The interfaces are either accessed directly by applications or over the network via a service endpoint. We explain both layers in more detail in the following sections.

4.4.1 Core Layer

The core layer is composed of four modules, namely *Uni Core*, *Query Core*, *Update Core*, and *TA Manager*. It implements the three basic operations described above for RDB-to-RDF translation and it is responsible for the interactions with the database system.

The *Uni Core* module provides the API of the core layer to the data access interface layer. It acts as a controller of the other three modules to manage the correct execution of requests including their encapsulation in database transactions. This API basically consists of two methods, one for query requests and one for update requests. The query method is a simple wrapper for the query method of the *Query Core* described below. The update method, on the other hand, takes as parameters one set of insert requests and one set of delete requests to execute all of them in the scope of a single database transaction (*e.g.*, for a SPARQL/Update request). Data access interfaces are required to collect requests that belong to a single transaction themselves and submit them all at once. This has the advantages that data access interfaces are relieved from managing transactions and the runtime of the transactions can be kept as short as possible. The *Uni Core* further isolates the *Query Core* and the *Update Core* from the database as it is responsible for collecting translated requests and for passing them to the database system.

The *TA Manager* module is responsible for database transaction management. It is used for starting, committing, and rolling back transactions based on instructions it receives from the *Uni Core* module.

The main parts of the core layer are the *Query Core* and *Update Core* modules that implement the RDB-to-RDF translation logic. In the following, we present them in more detail.

Query Core

The *Query Core* implements the basic operation of querying for an arbitrary triple pattern. Based on a given mapping, it translates the pattern to a single or multiple SQL queries depending on the type of pattern. For instance, a pattern asking for the object of a given subject and predicate will generate just a single SQL query whereas a pattern with variable subject and predicate but given object will result in multiple SQL queries as this object value could appear in multiple tables (and attributes) of the database.

Algorithms 3–6 illustrate this translation of triple patterns to SQL queries. First, we differentiate triple patterns that feature a concrete subject and such

Algorithm 3 translatePattern(subject, predicate, object)

```
1: if subject.isVariable() is false then
2:     table ← identifyTable(subject)
3:     queries ← translateTable(table, predicate, object)
4: else
5:     for all table in getTables() do
6:         queries ← translateTable(table, predicate, object)
7:     end for
8: end if
9: return queries
```

with a variable as subject (Algorithm 3). If the subject is concrete (*i.e.*, a resource URI) we use it to identify the table the pattern matches (line 2). On the other hand, a variable subject means that we potentially have to generate a SQL query for each mapped table (lines 5–7). Next, we check if the predicate of the pattern is concrete or a variable (Algorithm 4). A concrete predicate is translated to the matching attribute (line 2) while in the case of a variable predicate we can not know which attribute will match a given (object) value. We therefore have to incorporate all mapped attributes of a given table into the query (lines 5–6). Next, we differentiate concrete and variable objects (Algorithm 5). The database value is extracted from a concrete value according to the mapping definition (*e.g.*, extracting part of an URI). Otherwise, if the object is a variable we mark the value as *null* (line 4). Finally, we assemble the SQL query (line 6; Algorithm 6). The primary key of the affected table is added to the list of projected variables (*i.e.*, the *select* clause; line 1) and the table itself is added to the *from* clause. Then, we iterate over the attributes to add conditions to the *where* clause (lines 3–8). If the value is *null*, we add the attribute to the *select* list (lines 4–6) and a condition that this attribute must not be *null* to the *where* clause. Otherwise, a condition is added that states that the attribute must match the value. In any case, multiple conditions are combined using the *or* operator (line 7). Last, the query is built from the *select*, *from*, and *where* parts and returned (line 9).

After the translation, the resulting SQL queries are encapsulated in a `TripleIterator` that implements the `java.util.Iterator` interface to provide a standard means for iterating over the results of a triple pattern query. The triples are generated on demand from the results of the SQL queries, which are evaluated sequentially. At any moment there is only one active SQL query, this means in the beginning the first SQL query is evaluated and its result are used for generating the result triples. Only after this SQL `ResultSet` is exhausted the next SQL query is evaluated, and so on. This has two major advantages. First, it reduces the memory consumption as at any time only one SQL `ResultSet` object must be held in main memory, the remaining queries are stored as strings. Second, if the caller is not interested in all results (*e.g.*, only the first twenty result triples are of interest) it is possible that only a subset of the generated SQL queries need to be evaluated. In that case, this approach can also have a positive effect on performance by exploiting the given partitioning of the data into tables.

The `TripleIterator` is implemented as a look-ahead iterator, *i.e.*, it always generates the next triple in advance and caches it until `next()` is called. Then the cached triple is returned and the next one is generated and cached. This approach was taken due to the differences in the APIs of SQL `ResultSet` and Java `Iterator`. Iterators have a `hasNext()` method to check if there are any further results, the SQL API does not offer such a method. Instead, a boolean value is returned after moving the cursor to the next result that indicates if there are additional results. Look-ahead iterators

Algorithm 4 translateTable(table, predicate, object)

1: **if** $predicate.isVariable()$ is false **then**
2: $attribute \leftarrow identifyAttribute(table, predicate)$
3: $queries \leftarrow translateAttribute(table, attribute, object)$
4: **else**
5: $attributes \leftarrow getAttributes(table)$
6: $queries \leftarrow translateAttribute(table, attributes, object)$
7: **end if**
8: **return** $queries$

Algorithm 5 translateAttributes(table, attributes, object)

1: **if** $object.isVariable()$ is false **then**
2: $value \leftarrow extractValue(object)$
3: **else**
4: $value \leftarrow NULL$
5: **end if**
6: $queries \leftarrow assembleQuery(table, attributes, value)$
7: **return** $queries$

Algorithm 6 assembleQuery(table, attributes, value)

1: $select \leftarrow table.getPK()$
2: $from \leftarrow table$
3: **for all** $attribute$ in $attributes$ **do**
4: **if** $value == NULL$ **then**
5: $select \leftarrow attribute$
6: **end if**
7: $where \overset{OR}{\leftarrow} getCondition(attribute, value)$
8: **end for**
9: **return** $buildQuery(select, from, where)$

allow to bridge this API gap elegantly and with good performance by avoiding unnecessary movement of the result cursor.

Update Core

The *Update Core* implements the remaining two basic operations of adding and removing sets of triples. In either case, the triples are translated to (typically multiple) SQL DML statements. The translation is performed according to a generalized version of the algorithm presented in [Hert et al., 2010b] for translating SPARQL/Update *insert data* and *delete data* operations. We briefly recapitulate the basic idea of the algorithm and refer the reader to [Hert et al., 2010b] for more details. The translation for adding and removing triples is basically the same, the difference is only in the generated SQL statements (insert vs. delete). First, the triples are grouped into so-called subject groups based on equal subjects (*i.e.*, these triples have the same subject and therefore

affect the same record in the database). This allows us to translate each such group of triples individually. Second, the affected table is identified via the subject URI. In a third step, the mapping is used to check if the submitted triples satisfy certain integrity constraints of the database schema. Step four generates the SQL statement for adding or deleting this group of triples based on the mapping definition. The predicate of each triple is translated to an attribute of the affected table. The object is used as the data value either directly or if it is a resource URI by matching it against the template in the mapping and extracting the predefined substring. Each subject group is processed that way and the resulting SQL statements are collected. Finally, the statements are executed within a single database transaction to ensure the atomicity of the original request.

4.4.2 Data Access Interface Layer

The data access interface layer is responsible for establishing the connection between the core layer of ONTOACCESS and Semantic Web applications. This is realized with an extensible set of data access interfaces that are exposed to the applications either directly (*e.g.*, the Jena interface) or indirectly via the service endpoint described in the next section (*e.g.*, the Linked Data interface). The job of the individual interfaces is to translate the interface-specific operations to the basic ONTOACCESS operations and possible results back into the interface-specific format. The idea is that such interfaces are very lightweight and therefore simple to develop as the main translation work is performed in the core layer. Currently, the ONTOACCESS platform implements data access interfaces for multiple Semantic Web Frameworks (Jena, Sesame, RDF2Go), RDF Dump (for dumping all data as RDF to a file), Linked Data, ChangeSet, and a subset of SPARQL 1.1 that includes SPARQL/Update as proposed in [Seaborne et al., 2008].

As an example, we present the Jena data access interface in more detail. Jena [Carroll et al., 2004] uses a two-layer API to interact with RDF data. It is composed of the *Model* and the *Graph* APIs. The *Graph* API represents the

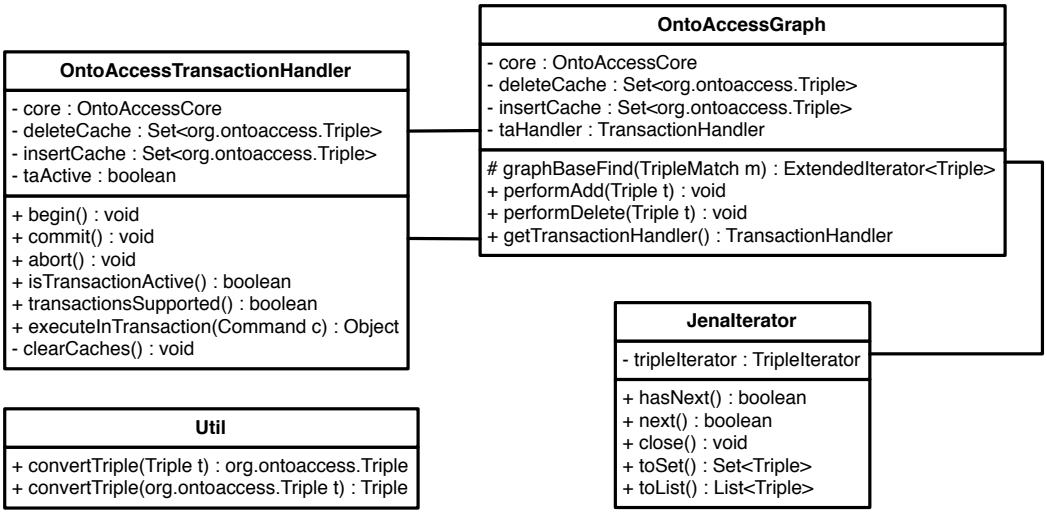

Figure 4.2: Jena Interface UML Class Diagram

lower layer and is responsible for retrieving and storing triples. It provides methods to add, delete, and find triples. The *Model* API is the layer facing the user and provides convenience methods for working with RDF data. These APIs are designed to be extended, *i.e.*, to add support for additional triple storage schemes. Jena itself provides multiple implementation for storing triples in memory or on disk. For the Jena interface of ONTOACCESS, we provide an implementation of the *Graph* API. Extending the *Model* API was not necessary as Jena contains a bridge to create a standard model from any implementation of the *Graph* API. Figure 4.2 depicts the UML class diagram of our Jena interface. It provides an implementation that interacts with the core layer of ONTOACCESS. The `OntoAccessGraph` class represents the main API class that contains the methods for adding, deleting, and finding triples. It maps them to the basic operations of ONTOACCESS. Further, the `JenaIterator` class implements the iterator interface that is returned from a call of the find method of the `OntoAccessGraph` class. It is basically a wrapper of our `TripleIterator` described in Section 4.4.1. Jena provides support for transactions by introducing dedicated transaction handler classes.

Our `OntoAccessTransactionHandler` implements this by collecting the triples intended to be added to or deleted from the data and forwarding them to the OntoAccess core to commit the transaction. The `Util` class contains convenience methods for converting between the triple representations of Jena and OntoAccess.

4.4.3 Service Endpoint

The *Service Endpoint* of OntoAccess is a server application that exposes data access interfaces as services to the network. Data access interfaces can be registered at the endpoint by providing the name of the implementing handler class and the HTTP request target (*e.g.*, `http://example.org/sparql?query=...` where *sparql* is the HTTP request target for the SPARQL service). If a request matches the target, its query string is forwarded to the respective data access interface for processing. Request strings that do not match any defined target are per default interpreted as Linked Data requests if the corresponding data access interface is installed or else as an error.

The service endpoint is implemented based on Java servlet technology[17] and the Jetty embedded Web server.[18]

4.5 Semantic Feedback Protocol

The conceptual gap between the relational model and RDF has a greater effect on translating RDF-based write requests to the database level than on read-only queries. If a query uses ontology terms (or instances) that cannot be mapped to the database schema (or data), the query can be processed without error but simply returns no results. However, if a write request contains such non-mappable ontology terms it cannot be fully processed and results in an error. Further, a write request can be underspecified *w.r.t.* the constraints defined in the database schema (*e.g.*, *not null* constraints). In such situations it

[17] `http://java.sun.com/products/servlet/`
[18] `http://jetty.codehaus.org/jetty/`

is possible to simply reject the whole request or ignore the parts that cannot be mapped. In both approaches, the client does not know why the request was not fully processed, especially if the client is unaware of the RDB-based foundation of the data storage system. Having said that, it cannot be expected (neither is it desirable) that all clients know about the specifics of the RDB schema if their usage is limited to the RDF-based data access interfaces. We therefore propose a semantic feedback protocol to alleviate this problem. The main idea of this feedback approach is to detect requests that are invalid *w.r.t.* the RDB schema already during request translation and in a second step provide feedback to the client in a semantic format such as RDF. In this way, there is no conceptual break between request and (error) response.

In ONTOACCESS, the semantic feedback protocol is implemented as a cross-layer feature. The detection of invalid write requests is performed in the *Update Core* where incoming requests are analyzed and translated. Requests are always fully analyzed to identify and report all invalid elements. The resulting feedback is stored in an internal format in the *Update Core* and it is exposed in a feedback interface to the *Data Access Interface Layer* and the *Service Endpoint*. The data access interfaces can implement the processing according to their needs (*e.g.*, the Jena interface could convert the feedback to Java exceptions). The *Service Endpoint* converts the feedback to an RDF-based format adhering to our semantic feedback ontology described later in this section. The feedback is published in this RDF-based format at the HTTP request target *feedback* of the *Service Endpoint* (*i.e.*, `http://example.org/feedback`) and can be retrieved with a simple *GET* request on that URL.

The remainder of this sections introduces the different types of feedback supported by our approach and our semantic feedback ontology. At last, we present a concrete example of feedback in the RDF-based format implemented in the *Service Endpoint*.

4.5.1 Feedback Types

We identified five causes for invalid write requests that can be detected during request translation. All of them arise from the conceptual gap between the relational model and RDF. We defined five corresponding feedback types which we present in detail.

MissingTriple

A *MissingTriple* feedback is generated if a request lacks data for a mandatory attribute in the RDB, *i.e.*, an attribute with a *not null* constraint. There are two cases that can lead to such a feedback. First, if data should be inserted that would create a new record in the database but the data of at least one mandatory attribute is missing in the request. Second, if data should be delete that corresponds to a subset of an existing record and includes data of a mandatory attribute. Both cases would lead to a database record with mandatory attributes set to *null* – a violation of constraints that would be prevented by the database management system. Requests that generate a *MissingTriple* feedback are always aborted.

UnknownSubject

An *UnknownSubject* feedback is generated if a request contains an RDF triple with a subject that cannot be mapped to a table of the RDB schema and can therefore not be stored. This type of feedback is exclusive to insert requests, because deleting a non-existing triple results in no operation on the data and can be silently ignored according to [Schenk et al., 2010]. It is a matter of configuration if requests that generate an *UnknownSubject* feedback should be aborted or continued without the affected triples. However, the feedback is always generated for information purposes.

UnknownTriple

An *UnknownTriple* feedback is generated if a request contains an RDF triple with a predicate that cannot be mapped to an attribute of the RDB schema and can therefore not be stored. In this feedback case the subject of the triple can be mapped to a table of the RDB schema or else it is classified as a *UnknownSubject* feedback as mentioned above. This type of feedback is exclusive to insert requests, because deleting a non-existing triple results in no operation on the data and can be silently ignored according to [Schenk et al., 2010]. It is a matter of configuration if requests that generate an *UnknownTriples* feedback should be aborted or continued without the affected triples. However, the feedback is always generated for information purposes.

NonMatchingTriple

A *NonMatchingTriple* feedback is generated if a request contains an RDF triple that can be mapped to the RDB schema but the value of the affected database record is already set and is different from the value of the object in the triple. This leads to an error because RDBs do not support storing multiple values for a single attribute. If the object value and the database value match it is not an error as inserting an already existing triple is silently ignored according to [Schenk et al., 2010]. In this feedback case the subject and predicate can be mapped to the RDB schema or else it would be classified as either a *UnknownSubject* or an *UnknownTriple* feedback as mentioned above. This type of feedback is exclusive to insert requests, because deleting a non-existing triple can be silently ignored [Schenk et al., 2010]. It is a matter of configuration if requests that generate an *NonMatchingTriple* feedback should be aborted or continued without the affected triples. However, the feedback is always generated for information purposes.

DefaultTripleAdded

A *DefaultTripleAdded* feedback is generated if a request lacks data for an attribute in the RDB that has a default value defined. Default values in RDBs

are silently added to new records if they are not provided by the client. The *DefaultTripleAdded* feedback informs the client about the additional data that was generated by translating it to an RDF triple. There are two cases that can lead to such a feedback. First, if data is inserted that would create a new record in the database but the data of at least one attribute with default value is not explicitly given in the request. Second, if data is deleted that includes the data of an attribute with default value. In this case the original data is deleted but the data is restored with the default value by the RDB system. The feedback is always generated for information purposes. Requests are never aborted because of a *DefaultTripleAdded* feedback.

An invalid write request may generate multiple feedback instances of the same or different types. The semantic feedback ontology in the next section is used to combine all feedback in a single feedback message for the client.

4.5.2 Semantic Feedback Ontology

The semantic feedback that is collected during the translation of write requests is provided to the client in an RDF-based format. This format is defined by our semantic feedback ontology described in this section. Table 4.1 presents an overview of the ontology with a list of all classes and short descriptions. The descriptions also include the most important ontology properties that are used with the respective class (*i.e.*, have that class as their *rdfs:domain*).

Examples

Listing 4.2 shows an example feedback document in the RDF-based format. It contains three individual feedback instances that we will explained in detail.

The first part of the feedback document is the definition of namespace prefixes as required by the Turtle RDF serialization [Beckett and Berners-Lee, 2011] (lines 1 to 5). Then, the main feedback message is listed (lines 7 to 11) that consists of the individual feedback instances (lines 8 to 10) and the date this request was processed (line 11). The rest of the document contains the three feedback instances. First, *fb:FB1* describes a *MissingTriple* feedback (line

Table 4.1: Semantic Feedback Ontology – Overview

Class	Description (including class specific properties)
FeedbackMessage	The collection of all feedback generated by a single write request. It lists all feedback instances (→ *hasFeedback*) that can be of any subclass of *FeedbackType*. It further contains the date (→ *dc:date*) this request was processed.
FeedbackType	The superclass for all types of feedback. It lists information about the severity of the feedback (→ *level: info, warn, error, fatal*), if it was generated during an *insert* or *delete* request (→ *source*), and if the request was *abort*ed or the source of the feedback was *ignore*d (→ *action*). The feedback further contains human readable descriptions of the feedback type (→ *rdfs:label* and → *rdfs:comment*). If the feedback affects a single triple, it is included in the feedback using RDF reification [Manola and Miller, 2004], *i.e.*, its subject (→ *rdf:subject*), predicate (→ *rdf:predicate*), and object (→ *rdf:object*) are listed.
MissingTriple	A subclass of *FeedbackType* that uses additional properties of the ontology. It lists information about the expected subject (→ *expectedSubject*), the expected predicate (→ *expectedPredicate*), and the expected datatype of the object value (→ *expectedObjectDatatype*) that the missing triple should be composed of.
UnknownSubject	A subclass of *FeedbackType* that uses an additional property of the ontology. It contains a list of triples (→ *triples*) that use the unknown subject.
UnknownTriple	A subclass of *FeedbackType* that uses no additional properties of the ontology.
NonMatchingTriple	A subclass of *FeedbackType* that uses no additional properties of the ontology.
DefaultTripleAdded	A subclass of *FeedbackType* that uses no additional properties of the ontology.
FeedbackAction	Indicates if the feedback resulted in aborting the request or not. The ontology defines two instances of this class named *Abort* and *Ignore*.
FeedbackLevel	Describes the severity of the feedback. The ontology defines four instances of this class named *Info*, *Warn*, *Error*, and *Fatal*. Depending on this severity level a client may react differently to the feedback. For instance, *Fatal* means the request was not executed at all while *Info* means it was successful, but some feedback was generated for information purposes.
FeedbackSource	Indicates if the feedback was generated during an insert or a delete request. The ontology defines two instances of this class named *Insert* and *Delete*.

```
@prefix  fb:    <http://ontoaccess.org/feedback/> .
@prefix  rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix  rdfs:  <http://www.w3.org/2000/01/rdf-schema#> .
@prefix  xsd:   <http://www.w3.org/2001/XMLSchema#> .
@prefix  dc:    <http://purl.org/dc/elements/1.1/> .

fb:FeedbackMessage1  a  fb:FeedbackMessage;
  fb:hasFeedback  fb:FB1,
                  fb:FB2,
                  fb:FB3;
  dc:date         "2011-10-05T13:37:21" .

fb:FB1  a  fb:MissingTriple;
  fb:action                 fb:Abort;
  fb:level                  fb:Fatal;
  fb:source                 fb:Insert;
  fb:expectedSubject        http://localhost:4040/uuc/Student3002;
  fb:expectedPredicate      http://xmlns.com/foaf/0.1/mbox;
  fb:expectedObjectDatatype  xsd:anyURI;
  rdfs:label                "MissingTriple";
  rdfs:comment              "A mandatory triple is missing ...".

fb:FB2  a  fb:DefaultTripleAdded, rdf:Statement;
  fb:action      fb:Ignore;
  fb:level       fb:Info;
  fb:source      fb:Insert;
  rdf:subject    http://localhost:4040/uuc/Student3002;
  rdf:predicate  http://ontoaccess.org/edu#grade;
  rdf:object     "1";
  rdfs:label     "DefaultTripleAdded";
  rdfs:comment   "A default triple was added ..." .

fb:FB3  a  fb:NonMatchingTriple, rdf:Statement;
  fb:action          fb:Abort;
  fb:level           fb:Error;
  fb:source          fb:Insert;
  rdf:subject        http://localhost:4040/uuc/Student1001;
  rdf:predicate      http://xmlns.com/foaf/0.1/firstName;
  rdf:object         "John";
  fb:expectedObject  "Bob";
  rdfs:label         "NonMatchingTriple";
  rdfs:comment       "A triple was detected ..." .
```

Listing 4.2: Semantic Feedback Example

13). As it is not possible to execute a request where mandatory triples are missing, the request was therefore aborted (line 14) leading to a severity of *fb:Fatal* (line 15). It can further be seen in the feedback that the request was an insert request (line 16). A *MissingTriple* feedback always includes information about what kind of triple was missing. The expected subject (line 17) is taken from the original request, the expected predicate (line 18) and the expected datatype of the object (line 19) are extracted from the mapping definition. At last, human readable descriptions of the feedback are given (lines 20 and 21; abbreviated). The second feedback is of type *DefaultTripleAdded* (line 23). This feedback would not result in the request being aborted, it is for information purposes (line 25) and could be ignored (line 24). The triple representation of the automatically added data is provided in the feedback as well using RDF reification [Manola and Miller, 2004] (lines 27 to 29). The final feedback for this request is a *NonMatchingTriple* feedback (line 33). It shows that it also leads to the the request being aborted (line 34) with a severity of *fb:Error* (line 35). This severity means that it would be possible to execute the request but this specific triple would not be stored. Instead the existing triple would remain valid. It is a matter of configuration if requests are aborted on feedback of severity *fb:Error*. The feedback contains the submitted triple (lines 37 to 39) as well as the object that exists already in the database (line 40).

Note that although each feedback instance has a *fb:action* property, a request is aborted if at least one of the feedback instances sets this property to *fb:Abort*.

4.6 Evaluation

The evaluation of our approach is split into three parts. First, in Section 4.6.1 we demonstrate by example how simple it is to extend the ONTOACCESS platform with data access interfaces. Second, in Section 4.6.2 we compare the performance of ONTOACCESS with D2R, Jena SDB,[19] an RDB-backed triple store, and Jena TDB,[20] a native RDF triple store. The benchmark experiment is

[19] `http://openjena.org/SDB/`
[20] `http://openjena.org/TDB/`

based on basic Jena API calls for querying, adding, and deleting triples since all evaluated approaches provide support for the Jena framework. Lastly, we summarize a case study we performed with OntoAccess in the domain of software analysis platforms.

4.6.1 Extensibility

In this paper, we claim that the OntoAccess platform provides simple extensibility to meet the requirement for developing additional data access interfaces. We will demonstrate this simple extensibility with three example implementations of data access interfaces, namely for *Jena*, *Linked Data*, and *ChangeSet*.

Jena

The Jena interface is an example that requires both read and write data access. Its implementation was already described in detail in Section 4.4.2, therefore we just add that it is one of the more complex interfaces and that it was implemented in about 300 lines of Java code.

Linked Data

The Linked Data interface is an example for read-only data access. It is one of the simpler interfaces and was implemented in about 100 lines of Java code. It provides support for linked data typed queries via the *Service Endpoint*. It takes an URI as input and returns all triples that have this URI as their subject. For that, it constructs a triple pattern with the given URI as subject and variables as predicate and object. This pattern is forwarded to the OntoAccess core for translation and evaluation. The resulting `TripleIterator` is wrapped in a `HtmlPartIterator` that emits the individual triples in HTML markup so that the *Service Endpoint* can directly stream the result page to the caller.

ChangeSet

The ChangeSet interface is an example for write-only data access. It is accessible via the *Service Endpoint* and it implements the ChangeSet protocol.[21] Its implementation was realized in about 200 lines of Java code and consists of the actual interface, the ChangeSet parser, and the implementation of the protocol. ChangeSet requests are RDF graphs adhering to the ChangeSet ontology.[22] They contain a so-called subject of change and two sets of matching triples. One triple set is meant for removal and the other for addition. Our ChangeSet interface implementation uses the Jena framework to parse the request and to extract the subject of change as well as the two triple sets. It then converts the parsed triples to the triple representation of ONTOACCESS and passes them via the *Uni Core* to the *Update Core* for addition and removal. To ensure the atomicity of a ChangeSet request, the addition and removal of the triples are executed within a single database transaction.

This brief description of implemented data access interfaces demonstrates how simple it is to develop such interfaces. It shows how few lines of code it requires compared to the translation logic in the core layer which consists of more than 8000 lines of Java code.

4.6.2 Performance

The layered architecture of the ONTOACCESS platform may suggest a disadvantage in performance compared to other RDB-to-RDF mapping approaches. In this section, we compare the performance of ONTOACCESS with D2R, Jena SDB, an RDB-backed triple store, and Jena TDB, a native RDF triple store. The benchmark experiment is based on basic Jena API calls for querying, adding, and deleting triples since all evaluated approaches provide support for the Jena framework. We show that ONTOACCESS delivers comparable or better performance than D2R and Jena SDB.

[21] `http://n2.talis.com/wiki/Changeset_Protocol`
[22] `http://purl.org/vocab/changeset`

Experimental Setup

The experiment was conducted on a Apple MacBook Pro notebook with a 2.33GHz Intel Core 2 Duo dual core CPU, 3GB of DDR2 667MHz RAM, a 320GB SATA HDD with 7200rpm running Mac OS X 10.6.3 as the operating system. As Java runtime we used version 1.6.0_17 provided with Mac OS X. As database system, MySQL version 5.1.45 was used for all systems with the default settings except `innodb_buffer_ pool_size` which was increased to 64MB via the *my.cnf* configuration file. The benchmark was run with a heap space allocation of 1024MB (*-Xmx1024m*).

We reused the dataset from the Berlin SPARQL Benchmark (BSBM) [Bizer and Schultz, 2009] in sizes equivalent to one million, ten millions, and hundred millions of triples. The datasets were generated with the BSBM data generator as described in the BSBM specification [Bizer and Schultz, 2008b]. The mapping for D2R was reused from a prior benchmark experiment conducted by the BSBM team. It is publicly available from their benchmark results website.[23] The mapping for ONTOACCESS was specifically developed for this evaluation. It is a complete mapping according to Definition 3.

The experiment consists of two parts, a query part and an update part. The query part tests the evaluation performance of single triple pattern queries. There exist eight such patterns including the one containing no variables (*i.e.*, a concrete triple) and the one containing only variables (*i.e.*, resulting in a dump of the database). The times reported below include the translation and evaluation of the queries as well as the retrieval of at most fifty result triples. The update part tests the performance of adding and deleting **A** a single triple, **B** a set of eight triples that affect a single table in the RDB, and **C** a set of thirteen triples that affect multiple tables. The results presented below were measured as the average of five benchmark runs after two warmup runs. The query part was executed for all systems under test (SUTs), the update part for ONTOACCESS, Jena SDB, and Jena TDB as D2R lacks support for data updates.

We used the following releases of the SUTs. ONTOACCESS in version

[23] `http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/results/`

0.3,[24] D2R in version 0.7, and Jena SDB in version 1.3.0 with the index-based database layout, and Jena TDB in version 0.8.4. All SUTs were used with default settings.

Results

Table 1 depicts the results of the query benchmark for datasets equivalent to one, ten, and hundred millions of triples. The first column names the approach and the dataset size. The remaining eight columns show the benchmark result times in milliseconds for each of the eight possible triple patterns. The triple patterns are depicted as a combination of the letters *s*, *p*, *o* that represent concrete subjects, predicates, objects and the question mark *?* that represents variables. For instance, the triple pattern *(s p ?)* represents a pattern with concrete subject and predicate but variable object.

Table 4.2: Result Times for Query Benchmark [ms]

1M	**(s p o)**	**(s p ?)**	**(s ? o)**	**(s ? ?)**	**(? p o)**	**(? p ?)**	**(? ? o)**	**(? ? ?)**
OntoAccess	1	1	2	2	5	8	724	39
D2R	2	2	7	5	8	9	267	110
JenaSDB	16	11	6	13	5	71	4	36 427
JenaTDB	12	9	13	16	17	23	13	2
10M	**(s p o)**	**(s p ?)**	**(s ? o)**	**(s ? ?)**	**(? p o)**	**(? p ?)**	**(? ? o)**	**(? ? ?)**
OntoAccess	1	1	3	4	203	224	216	489
D2R	2	2	7	6	202	235	1 051	1 369
JenaSDB	34	21	26	33	25	520	20	–*
JenaTDB	24	24	31	28	93	10	87	2
100M	**(s p o)**	**(s p ?)**	**(s ? o)**	**(s ? ?)**	**(? p o)**	**(? p ?)**	**(? ? o)**	**(? ? ?)**
OntoAccess	2	1	3	4	1 962	2 222	1 952	5 130
D2R	2	2	8	7	1 998	2 341	3 650	13 725
JenaSDB	42	30	45	60	70	5 320	100	–*
JenaTDB	38	32	53	43	316	10	99	3

* *crashed with a* `java.lang.OutOfMemoryError`

[24] available for download at `http://ontoaccess.org/`

The results show that for triple patterns with known subject, OntoAccess performs comparable to D2R and better than Jena SDB and Jena TDB. For triple patterns with unknown subject the results are mixed. Compared to D2R, OntoAccess performs comparable for the patterns with known predicate and in general better for the patterns with unknown predicate. The performance of Jena TDB for triple patterns with unknown subject is similar to patterns with known subject and therefore better than OntoAccess. Jena SDB performs better than OntoAccess for the patterns *(? p o)* and *(? ? o)* but worse for the other two. The performance of evaluating patterns with known predicate could be improved in OntoAccess and D2R if a database index is created on the attribute that is mapped to the property p. Tests showed that this reduces evaluation times to the levels of triple patterns with known subject. Also note that Jena SDB is only able to evaluate the *(? ? ?)* pattern in the one million triple dataset. It crashes with a `java.lang.OutOfMemoryError` error in larger datasets even if the heap memory allocation is doubled to 2048MB.

Table 2 depicts the results of the update benchmark for datasets equivalent to one, ten, and hundred millions of triples. The first column names again the approach and dataset size. The remaining six columns show the benchmark result times in milliseconds for adding and removing the three different triple sets. **A** represents the single triple, **B** the set of eight triples affecting a single table in the RDB, and **C** the set of thirteen triples affecting multiple tables. We report results for OntoAccess, Jena SDB, and Jena TDB as D2R is limited to read-only queries.

The results show that OntoAccess performs better than Jena SDB in adding and removing triples irrespective of the triple set or dataset size. The performance difference is especially striking in the removal of triples. A closer examination revealed that Jena SDB translates the removing of each individual triple to a SQL statement that needs to perform multiple joins on large tables. OntoAccess, on the other hand, translates the removing of triples to a single, join-less SQL statement for each affected table. Compared to Jena TDB OntoAccess performs better on the two larger data sets and the performance difference increases with the number of triples to add or remove.

Table 4.3: Result Times for Update Benchmark [ms]

1M	add A	remove A	add B	remove B	add C	remove C
OntoAccess	4	3	5	4	6	8
JenaSDB	11	1 311	42	10 328	60	16 776
JenaTDB	2	3	10	8	12	12
10M	**add A**	**remove A**	**add B**	**remove B**	**add C**	**remove C**
OntoAccess	5	3	5	5	6	8
JenaSDB	11	13 601	44	111 150	78	180 532
JenaTDB	9	4	47	13	98	17
100M	**add A**	**remove A**	**add B**	**remove B**	**add C**	**remove C**
OntoAccess	4	3	4	5	5	7
JenaSDB	30	329 184	198	2 676 356	238	4 356 603
JenaTDB	15	3	77	13	134	16

Note: D2R is missing from this table as it is limited to read-only queries

4.6.3 Case Study

In [Hert et al., 2011a], we presented a case study on how ONTOACCESS can be used to facilitate the transition from legacy systems to Semantic Web-enabled applications in practice. The case study showed how we successfully used ONTOACCESS to advance our Eclipse-based software evolution analysis framework EVOLIZER [Gall et al., 2009] to SOFAS [Ghezzi and Gall, 2011], a service-oriented, distributed, and collaborative software analysis platform. To motivate our case study, we present use cases that require interoperability between EVOLIZER and SOFAS. These use cases need a bidirectional data exchange, *i.e.*, from EVOLIZER to SOFAS and vice versa. First, EVOLIZER contains data about the software life-cycle of hundreds of software systems. Re-importing this vast amount of data in SOFAS from version control and bug tracking systems would take months, and some of these repositories might not even be available online anymore. Therefore, RDF-based *read access* to the EVOLIZER database is needed. Second, EVOLIZER implements importers to import source code and history data from centralized version control systems, such as CVS and SVN. Lately decentralized version control systems, such

as Git or Mercurial, gained popularity. Therefore, respective import services were developed for the SOFAS platform. The data produced by these importer services is modeled in RDF. It would also be valuable to EVOLIZER because existing tools could be used to leverage it. This, however, requires RDF-based *write access* to the EVOLIZER database. Lastly, SOFAS implements an extensible framework to compute software metrics on the data. Again, this data is modeled in RDF, but matching relations are available in the EVOLIZER database schema. RDF-based *write access* to the RDB is needed to make the metrics data available to EVOLIZER. These use cases indicate that, for making a bridge between EVOLIZER and SOFAS, an RDB-to-RDF mapper such as ONTOACCESS is needed that provides RDF-based read and write access to RDBs.

4.7 Conclusion

In this paper, we presented ONTOACCESS as an extensible platform for RDF-based read and write access to data stored in existing RDBs. We discussed that there are many different data access approaches in current Semantic Web applications and that a platform-based approach is needed to avoid repeated implementation effort in RDB-to-RDF translation. We identified three basic operations that such a platform has to provide in its core implementation, namely (1) querying for a single triple pattern, (2) adding triples, and (3) removing triples. These basic operations are implemented in the core layer of ONTOACCESS and we discussed that this architectural decision enables the simple implementation of various data access interfaces in the interface layer.

We introduced a semantic feedback protocol to bridge the conceptual gap between the relational model and RDF. It informs the client about invalid write request in an RDF-based format and provides recommendation on how to change the request for the better. We presented the semantic feedback ontology and the implementation in ONTOACCESS.

We showed that this platform-based approach performs comparable or better than existing read-only RDB-to-RDF mapping approaches as well as current RDB-based triple stores.

We further introduced a formal definition of our RDB-to-RDF mapping and proofs of its bidirectional properties. The rationale of the formal definition and the proofs is to show that our mapping language R3M is bidirectional and therefore not affected by the view update problem.

5

How to "Make a Bridge to the New Town" using OntoAccess

Matthias Hert, Giacomo Ghezzi, Michael Würsch, Harald C. Gall
Published at the 10th International Semantic Web Conference, 2011
DOI: 10.1007/978-3-642-25093-4_8

Abstract

Business-critical legacy applications often rely on relational databases to sustain daily operations. Introducing Semantic Web technology in newly developed systems is often difficult, as these systems need to run in tandem with their predecessors and cooperatively read and update existing data.

A common pattern is to incrementally migrate data from a legacy system to its successor by running the new system in parallel, with a data bridge in between. Existing approaches that can be deployed as a data bridge in theory,

restrict Semantic Web-enabled applications to read legacy data in practice, disallowing update operations completely.

This paper explains how our RDB-to-RDF platform ONTOACCESS can be used to transition legacy systems into Semantic Web-enabled applications. By means of a case study, we exemplify how we successfully made a bridge between one of our own large-scale legacy systems and its long-term replacement. We elaborate on challenges we faced during the migration process and how we were able to overcome them.

5.1 Introduction

The field of software engineering is in a constant state of flux. New paradigms, programming languages, frameworks, and tools gain tremendous momentum all of a sudden – and then they sink into oblivion as quickly as they have emerged. Short time-to-marked intervals are therefore critical for the success of new tools, be it in an industrial context or in research.

In contrast to short-lived tools, the body of acquired knowledge of a company usually evolves less rapidly and sometimes even remains relevant for decades, stored as data in different, mostly relational databases (RDB). This inevitably leads to challenges, when different generations of applications have to operate on this data.

There are legacy systems relying on a relational view of the database—these applications can not easily be upgraded or simply taken offline and thrown away when requirements change. Legacy systems are often crucial for daily operations and therefore need to be highly available. They are inherently valuable to many organizations but bear typical problems: Maintenance and especially further development have become difficult and costly.

These circumstances and new business opportunities emerging with the advent of paradigms, such as Service-Oriented Architectures and the Semantic Web, lead to the development of next-generation systems. While new development opens the door for incorporating recent best-practices and state-of-the-art technologies, the newly developed applications usually will run in tandem with legacy systems and still need to access legacy databases.

In such scenarios, it is common to *make a bridge to the new town*, that is, to incrementally migrate data from a legacy system by running the new system in parallel, with a data bridge in between [Demeyer et al., 2002]. Tools such as D2R Server [Bizer and Cyganiak, 2006] and OpenLink Virtuoso [Erling and Mikhailov, 2007] serve RDF views on relational databases. However, they restrict Semantic Web-enabled applications to read legacy data, disallowing update operations completely.

In this paper, we describe how our RDB-to-RDF platform ONTOACCESS [Hert, 2009] can be used to facilitate the transition from legacy systems to Semantic Web-enabled applications in practice. ONTOACCESS provides a semantic layer on top of existing relational databases. It enables RDF-based read and write access to relational data. Based on mappings that bridge the conceptual gap between RDF and the relational model, a mediator translates Semantic Web requests on-the-fly to SQL. This enables relational and RDF-based applications to cooperate on the same data and to further exploit the advantages of the well established database technology such as query performance, scalability, transaction support, and security.

The contribution of this paper is a case study on how we successfully used ONTOACCESS to advance our Eclipse-based software evolution analysis framework EVOLIZER [Gall et al., 2009] to SOFAS [Ghezzi and Gall, 2011], a service-oriented, distributed, and collaborative software analysis platform. We describe use cases where existing RDB-to-RDF approaches are insufficient and an approach such as ONTOACCESS is needed.

The remainder of this paper is structured as follows. Section 5.2 gives a brief introduction to the two systems between which ONTOACCESS acts as a data-bridge: EVOLIZER and SOFAS. ONTOACCESS itself bridges the conceptual gap between the relational model and RDF. It is described in Section 5.3. In Section 5.4, we present the case study on how we successfully used ONTOACCESS to advance EVOLIZER to SOFAS. Related work in the context of RDB-to-RDF mapping is reviewed in Section 5.5. Section 5.6 concludes this paper with a summary.

5.2 Background

In this section, we describe our two platforms for software analysis that run in tandem and are able to share data thanks to ONTOACCESS. The first platform, EVOLIZER, is considered to be a legacy system, whereas SOFAS represents our latest ambitions in providing a scalable, distributed means to analyze the evolution of a software system.

5.2.1 Evolizer

In the past, we have developed EVOLIZER [Gall et al., 2009] – a plug-in-based software evolution analysis and research platform, tightly woven into the Eclipse IDE.

At its core, EVOLIZER is based on the idea of a *Release History Database (RHDB)* [Fischer et al., 2003]. It is implemented as a set of Eclipse plug-ins and integrates information originating from different software repositories, such as version control, issue tracking, mailing lists, etc. The combination of this diverse, yet interconnected data allows one to uncover and analyze the many different facets of the evolution of a software system and its parts. Examples are the system's fine-grained change history or bug-proneness over time, as well as a complete source code model.

EVOLIZER has become a typical legacy system over time: While the platform is still in active use, it becomes harder to adapt it to new requirements and recent advances in technology. The tight coupling to Eclipse makes it hard to adapt and re-use EVOLIZER's tools and algorithms in new environments such as in a service-oriented context. Further, the RHDB is based on classical relational database technology. It is therefore difficult to interlink information stored in the RHDB with other external data sources, because the relations that we store are local – not universal – and our entities lack unique resource identifiers that can be dereferenced over the Internet. Synergies with related approaches are therefore difficult to exploit. EVOLIZER's models also lack explicit semantics, such as cardinality, transitivity, symmetry, and so on. Bringing EVOLIZER and its RHDB to the Semantic Web would therefore be desirable.

EVOLIZER is still very valuable and our RHDB contains data about the software life-cycle of hundreds of systems. Re-importing this vast amount of data from version control and bug tracking systems would take months, and some of these repositories might not even be available online anymore.

To overcome these limitations, we are in need of a gradual migration path from EVOLIZER to the next generation of software evolution analysis platforms, allowing us to run the existing platform together with its replacement for the years to come.

5.2.2 SOFAS

Evolizer allowed us to combine and analyze different aspects of a software's evolution and its development. However, we realized that a big potential lies in having analyses easily accessible and composable, without platform and language limitations, and not having to install and configure particular tools. Based on these premises, we introduced the concept of "Software Analysis as a Service" [Ghezzi and Gall, 2008]: getting easy access to different analyses from various tools and providers using Web services. We implemented that concept into a lightweight and flexible platform called SOFAS (SOFtware Analysis Services) [Ghezzi and Gall, 2011].

SOFAS follows the principles of a RESTful architecture [Fielding, 2000] and allows for a simple yet effective provisioning and use of software analyses based upon the principles of Representational State Transfer around resources on the Web. Its architecture is made up by three main constituents: Software Analysis Web Services, a Software Analysis Broker, and Software Analysis Ontologies. The services expose their functionalities and data through standard RESTful Web service interfaces. The Software Analysis Broker acts as the service manager and provides the interface between the services and the users. It contains a catalog of all the registered analysis services with respect to a specific software analysis taxonomy. As such, the domain of analysis services is described in a semantic way, enabling users to browse and search for their analysis service of interest. The ontologies – we call them SEON (*cf.*

Section 5.4.2) – are used to define and represent the RDF data consumed and produced by the different services.

5.3 OntoAccess as a Bridge to the New Town

ONTOACCESS is an RDB-to-RDF mediation platform that enables Semantic Web-based applications to operate on relational data. It provides a semantic layer on top of existing relational databases to enable RDF-based read and write access to the relational data. Semantic Web requests, *i.e.*, query and update requests, are translated on-the-fly to SQL for execution in the database. ONTOACCESS therefore eliminates the need for mirroring and synchronizing the relational data and the RDF representation – both data models always operate on the same state of the data. This results in a cooperative use of the data in RDF-based as well as in relational applications. In addition, mediation allows one to further exploit the advantages of the well-established database technology such as query performance, scalability, transaction support, and security. The existing, read-only RDB-to-RDF mapping approaches are limited to data warehouse-like applications where the data can be queried and analyzed but not modified. In comparison, ONTOACCESS puts relational databases on par with native RDF triple stores by allowing read and write access to the data. This facilitates the transition from RDB-based legacy systems to Semantic Web-enabled applications in practice.

5.3.1 Architectural Principles of OntoAccess

ONTOACCESS is designed and implemented as an extensible platform. It encapsulates the RDB-to-RDF translation logic in the core layer which provides the foundation for an extensible set of data access interfaces in the interface layer. The RDB-to-RDF core is responsible for the translation of RDF-based request to SQL and interacts with the database system. The interface layer exposes the functionality of the ONTOACCESS core to RDF-based applications via different data access approaches. It translates the interface-specific

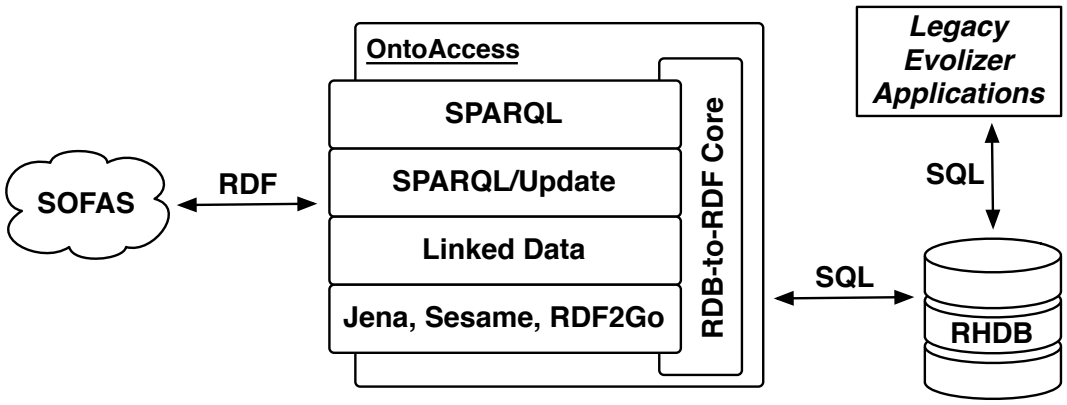

Figure 5.1: OntoAccess Architecture Overview

operations to the basic ONTOACCESS operations, and results back into the interface-specific format. This facilitates the development of additional data access interfaces because the main RDB-to-RDF translation work is performed in the core layer. Currently, the ONTOACCESS platform supports data access via SPARQL [Prud'hommeaux and Seaborne, 2008], SPARQL/Update [Seaborne et al., 2008], Linked Data [Berners-Lee, 2009a], and various Semantic Web Frameworks, such as Jena, Sesame, and RDF2Go.[1] The data access interfaces are accessible via a HTTP service endpoint if deployed as a stand-alone server. Alternatively, ONTOACCESS can be integrated into other applications as a library, in which case the data access interfaces are exposed via specific APIs. Figure 5.1 presents an overview of the overall architecture of ONTOACCESS and exemplifies how it can be used as a data-bridge in the context of SOFAS and the EVOLIZER RHDB. Next, before we discuss this example in detail in Section 5.4, we elaborate on the mapping principles of ONTOACCESS.

[1] `http://openjena.org/`, `http://openrdf.org/`, `http://rdf2go.semweb4j.org/`

5.3.2 Mapping Principles of OntoAccess

Mediation requires a mapping from concepts in a relational database schema to terms defined in an ontology. For ONTOACCESS, we developed R3M [Hert et al., 2010b] as a bidirectional RDB-to-RDF mapping language that incorporates the requirements of RDF-based write access to relational databases. Existing mapping languages developed for read-only use cases are unsuitable for write access as shown in [Garrote and Garcia, 2011] and by D2R/Update.[2]

R3M extends the mapping approach described in [Berners-Lee, 2009b]. Tables of the database schema are mapped to classes in an ontology and the attributes of those tables to properties. Special support is provided for *link tables* that are used to represent M:N relationships in the relational model. As such helper constructs are not needed in RDF, link tables are mapped to properties instead of classes. In addition, R3M mappings contain information about datatypes, as well as integrity constraints of the database schema. This results in a mapping language that is not as expressive as the existing, read-only languages (*cf.* [Hert et al., 2011c]) but it is sufficient to cover many application scenarios, including the one presented in this paper. In general, R3M is targeted at mapping highly normalized relational database schemata such as the ones generated by object-relational mappers (*e.g.*, Hibernate[3]), and at the so-called *direct mapping* where an equivalent RDF representation of the relational data is needed (*e.g.*, for use cases as described in [Fürber and Hepp, 2010]).

Listing 5.1 presents examples of the three main mapping constructs in R3M. The namespace prefixes used in the examples are defined as follows: `r3m` represents our mapping language vocabulary `http://ontoaccess.org/r3m/` while `ex` is used for the namespace `http://example.com/mapping/` of our example mapping. `ver` represents the namespace of the SEON version control ontology `http://evolizer.org/ontologies/seon/2010/03/versions.owl` (*cf.* Section 5.4.2). Listing 5.1a) depicts a *TableMap* representing the mapping of a database table to a class in the ontology. A *TableMap* con-

[2] `http://d2rqupdate.cs.technion.ac.il/`
[3] `http://hibernate.org/`

```
1  a)  ex:revision  a  r3m:TableMap;
2        r3m:hasTableName  "Revision";
3        r3m:mapsToClass   ver:Version;
4        r3m:uriPattern    "http://.../revision_%%number%%";
5        r3m:hasAttribute  ex:revision_number, ....
6
7  b)  ex:revision_number  a  r3m:AttributeMap;
8        r3m:hasAttributeName      "number";
9        r3m:mapsToObjectProperty  ver:hasID;
10       r3m:dbType                [ a  r3m:VarChar;
11                                   r3m:length 255 ];
12       r3m:hasConstraint         [ a  r3m:NotNull ].
13
14 c)  ex:release_revision  a  r3m:LinkTableMap;
15       r3m:hasTableName          "Release_Revision";
16       r3m:mapsToObjectProperty  ver:comprises;
17       r3m:hasSubjectAttribute   ex:rr_release;
18       r3m:hasObjectAttribute    ex:rr_revision.
```

Listing 5.1: Example R3M Mappings

tains the name of the table (line 2) and the class it is mapped to (line 3). The URI pattern (line 4, abbreviated) is used to generate the URIs for instances of this table based on values of table attributes that are specified between double percentage signs (e.g. %%*number*%% where *number* is the name of an unique attribute such as the primary key). A *TableMap* further contains a list of *AttributeMap*s (line 5, abbreviated).

Listing 5.1b) presents an example of an *AttributeMap* that maps a database attribute to a property in the ontology. An *AttributeMap* contains the name of the attribute in the database schema (line 8) and the property it is mapped to (line 9). Additionally, an *AttributeMap* includes information about the datatype of the database attribute (lines 10 and 11) as well as information about (database) constraints defined on that attribute (e.g., a not null constraint; line 12). R3M supports the constraints `r3m:PrimaryKey`, `r3m:ForeignKey`, `r3m:NotNull`, `r3m:Default`, and `r3m:Check`.

Listing 5.1c) shows a *LinkTableMap* representing the mapping of a link table to an ontology property. A *LinkTableMap* specifies the name of the link table in the database (line 15) and the property it is mapped to (line 16). A link table always contains two foreign key attributes that point to the tables of the N:M relationship. These attributes are represented as *AttributeMap*s (line 17 and 18) that provide the names of the attributes, the foreign key references to the tables, and the direction of the relationship (from subject to object).

5.4 A Case Study on Bridging Software Analysis Data

To motivate our case study, we present use cases that require interoperability between EVOLIZER and SOFAS. These use cases need a bidirectional data exchange, *i.e.*, from EVOLIZER to SOFAS and vice versa.

First, EVOLIZER contains data about hundreds of software systems that were imported over the past years (*cf.* Section 5.2.1). The SOFAS platform needs to be able to access this data without the need for re-importing it. This requires RDF-based **read access** to the EVOLIZER database. Second, EVOLIZER implements importers to import source code and history data from centralized version control systems, such as CVS and SVN. Lately decentralized version control systems, such as Git or Mercurial, gained popularity. Therefore, respective import services were developed for the SOFAS platform. The data produced by these importer services is modeled in RDF, based on the SEON ontologies described in Section 5.4.2. This data is also valuable to EVOLIZER because existing tools could be used to leverage it. This, however, requires RDF-based **write access** to the EVOLIZER database. Lastly, SOFAS implements an extensible framework to compute software metrics on the data. Again, this data is modeled in RDF, but matching relations are available in the EVOLIZER database schema. RDF-based write access to the RHDB is needed to make the metrics data available to EVOLIZER.

These use cases indicate that, for making a bridge between EVOLIZER and SOFAS, an RDB-to-RDF mapper is needed that provides RDF-based read and write access to relational data. Whereas existing approaches are limited to read-only queries, we developed ONTOACCESS that provides read and write access. In the remainder of this section, we present the relational data model of EVOLIZER, the ontology-based data model of SOFAS, and how the mapping of ONTOACCESS is used to bridge the conceptual gap between these two data models. We further discuss challenges we encountered during this case study.

5.4.1 Data Schema of Software Analysis within Evolizer

The data schema of EVOLIZER consists of several distinct parts covering many aspects of the Software Engineering domain. For this case study, we focus on those parts that are concerned with historical aspects and with source code information. The history model is generic, *i.e.*, it applies to a certain extent to centralized version control systems, as well as to distributed ones. In the following, we describe the most important parts of our schema. An overview of the simplified version of the schema is given in Figure 5.2; the full schema consists of approximately 40 distinct tables.

One of the core entities is *Revision*. A revision is a particular version of a file; a *Person*, that is to say a software developer, edits a file, and commits the modifications to the version control system. The latter tracks the date of the commit, the reason for the modification (*i.e.*, the commit message provided by the developer), as well as additional information such as the number of lines that were affected. A *Release* constitutes an important milestone in the life-cycle of a software system. It is often identified by a codename and contains a snapshot of the most recent revisions of all the files at the release data. New or experimental features, as well as bug fixes, are often developed on a *Branch*. Once the code is stable, it is merged back into the trunk.

To obtain a meaningful source code model, the system under analysis needs to be in a consistent state. This is generally guaranteed only for a release and therefore, for revisions that are part of a release, we can parse the source code

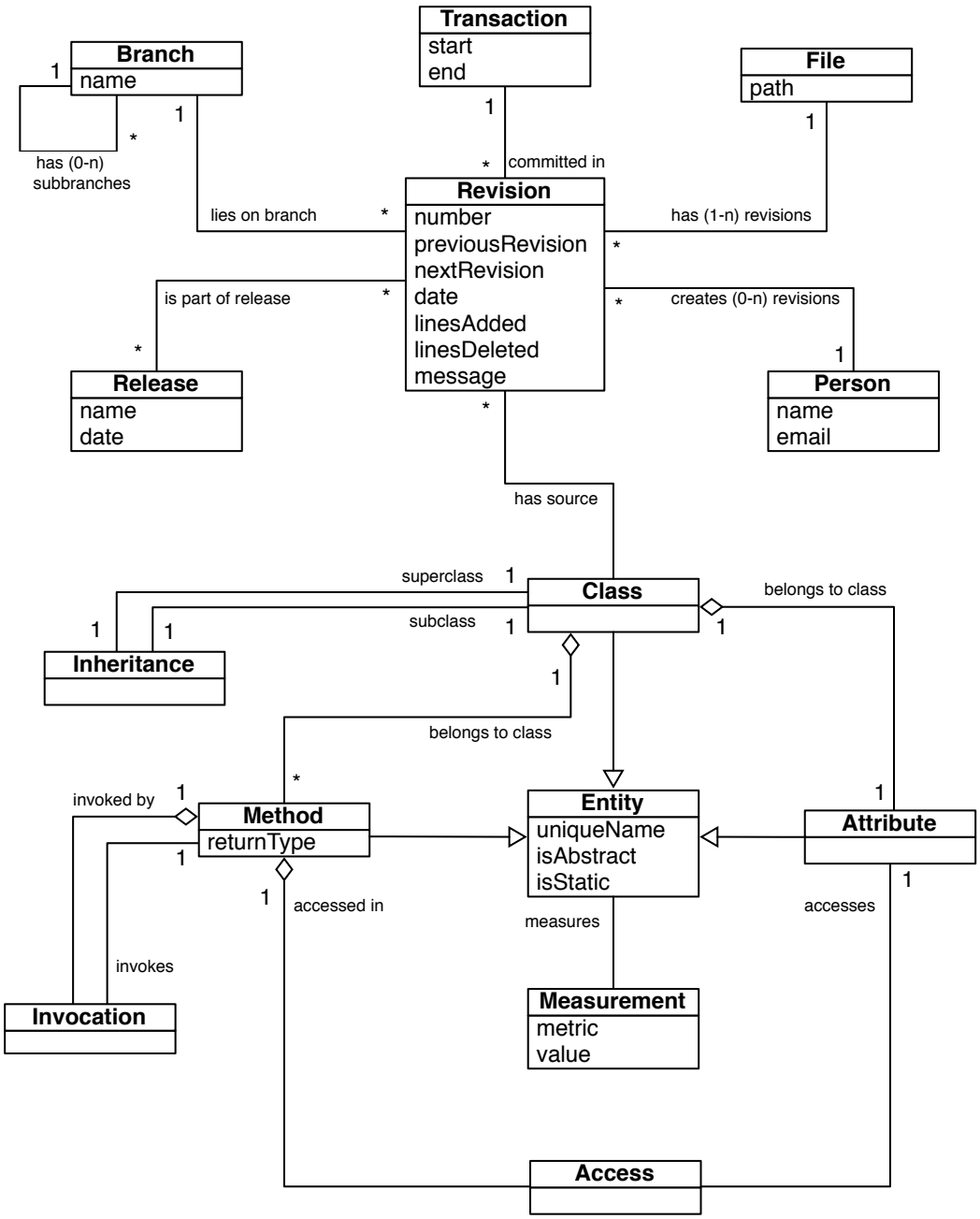

Figure 5.2: Evolizer Data Schema for Source Code and Historical Analysis

and instantiate a reasonable model accordingly. A revision then corresponds to one top-level *Class* in Java, C#, etc. Classes have a set of members, *i.e.*, they contain *Attributes* and *Methods*. Classes, attributes, and methods are generalized into *Entities*.

Relationships between source code entities, such as *Invocations* between methods, *Accesses* from methods to attributes, and *Inheritance* between classes, are also made explicit by representing them as an association class or link table. While they are hard to distinguish from real entities, this is the only means that the relational model provides when we want to explicitly query for relationships.

Entities can be measured. Such a *Measurement* is specified by a metric, for example 'number of lines of code' for a class or method, or 'number of accesses' for an attribute, and the value that has been measured.

5.4.2 Ontologies of Software Analysis within SOFAS

To describe the data produced and consumed by SOFAS, we developed a family of Software Evolution ONtologies (SEON). They describe different aspects of software and its evolution, such as version control, issue tracking, static source code structure, change coupling, software design metrics, and so on. SEON is organized as several ontology pyramids. For each of the major subdomains, we have developed higher level ontologies defining their common concepts. For system-specific or language-dependent concepts we developed some concrete low-level ontologies. For instance, there is a high-level version control ontology and several low-level ontologies for concrete version control systems, such as CVS, SVN, and Git, that extend the high-level version control ontology. In this paper, we limit the discussion to the main terms of the source code ontology and the version control ontology. The source code ontology models the static source code structures based on the FAMIX meta model. It is therefore similar to the EVOLIZER data schema described in the previous section. Table 5.1 provides an overview of the main classes and properties of the SEON source code ontology. The full ontology covers many more concepts such as *interfaces, local variables,* and *exceptions*.

Table 5.1: Source Code Ontology Overview

Class: Class	**Class: Method**
→ declaresMethod : Method	→ accessesField : Field
→ declaresField : Field	→ hasParameter : Parameter
→ isReturnTypeOf : Method	→ invokesMethod : Method
→ isSubclassOf : Class	→ hasReturnClass : Class
→ isSuperclassOf : Class	→ isInvokedByMethod : Method
→ hasName : xsd:string	→ isMethodOf : Class
Class: Field	→ hasName : xsd:string
→ isDeclaredFieldOf : Class	**Class: Parameter**
→ isAccessedByMethod : Method	→ isParameterOf : Method
→ hasName : xsd:string	→ hasName : xsd:string

Table 5.2: Version Control Ontology Overview

Class: Version	**Class: ChangeSet**
→ hasID : xsd:string	→ hasCommitDate : xsd:date
→ follows : Version	**Class: Branch**
→ precedes : Version	→ hasTag : xsd:string
→ hasCreationDate : xsd:date	**Class: Release**
→ linesAdded : xsd:int	→ hasReleaseDate : xsd:string
→ linesDeleted : xsd:int	→ hasTag : xsd:string
→ hasMessage : xsd:string	

The version control ontology models the structure of version control systems and is based on the data model described in [Fischer et al., 2003]. Table 5.2 provides an overview of the main classes and properties of the SEON version control ontology.

5.4.3 OntoAccess as a Bridge to the New Town of Software Analysis

ONTOACCESS bridges the conceptual gap between the RDB-based EVOLIZER and the Semantic Web-enabled SOFAS. It introduces an RDB-to-RDF mapping

and provides on-the-fly translation of RDF-based read and write requests to the EVOLIZER RHDB. Table 5.3 and Table 5.4 contain an overview of the mapping in a schematic representation. Again, we focus in the presentation of the mapping on the parts of the EVOLIZER RHDB that are relevant to this case study. The mapping uses the following namespace declarations. `ver` for the SEON version control ontology `http://evolizer.org/ontologies/seon/2010/03/versions.owl`, `top` for `http://evolizer.org/ontologies/seon/2010/03/top.owl`, `java` for the SEON source code ontology `http://evolizer.org/ontologies/seon/2009/06/java.owl`, and `foaf` for `http://xmlns.com/foaf/0.1/`. Table 5.3 lists the mapping of tables from Figure 5.2 that represent a domain concept and their attributes. The table consists of four columns. The first names the table as in Figure 5.2 and the second the class in the ontology it is mapped to. Column 3 contains the attributes of the respective table and their mapping to properties depicted in Column 4. A dash in the Columns 2 or 4 means that there is no mapping. The table *Entity* is not mapped to a class in the ontology but its attributes are mapped to properties. *Entity* is just a super type of several of the other concepts and only those (sub-)concepts are represented in the ontology (*cf.* Section 5.4.4).

Table 5.4 lists the mapping of link tables that represent M:N relationships in RDBs. As RDF provides different means to represent M:N relationships, such helper constructs are not needed and those tables are mapped to ontology properties instead. The table consists of three columns. The first names the link tables that are represented in Figure 5.2 as connecting lines between two concepts or as explicit concepts themselves. In the first case, the name is composed of the two participating table names separated by an underscore. Column 2 lists the property that each link table is mapped to. Column 3 lists the corresponding inverse property. For instance, the relationship from *Release* to *Revision* is mapped to the property `ver:comprises` and the inverse relationship from *Revision* to *Release* is mapped to the property `ver:appearsIn`.

Table 5.3: Mapping Overview Part I

table	→	class	attribute	→	property
Revision	→	`ver:Version`	number	→	`ver:hasID`
			previousRevision	→	`ver:follows`
			nextRevision	→	`ver:precedes`
			date	→	`ver:hasCreationDate`
			linesAdded	→	`ver:linesAdded`
			linesDeleted	→	`ver:linesDeleted`
			message	→	`ver:hasMessage`
Transaction	→	`ver:ChangeSet`	start	→	-
			end	→	`ver:hasCommitDate`
Branch	→	`ver:Branch`	name	→	`ver:hasTag`
File	→	`top:File`	path	→	`top:filePath`
Release	→	`ver:Release`	name	→	`ver:hasTag`
			date	→	`ver:hasReleaseDate`
Person	→	`foaf:Person`	name	→	`foaf:name`
			email	→	`foaf:mbox`
Entity	→	–	isAbstract	→	`java:isAbstract`
			isStatic	→	`java:isStatic`
Class	→	`java:Class`			
Method	→	`java:Method`	returnType	→	`java:hasReturnType`
Attribute	→	`java:Field`			
Measurement	→	`met:Metric`	metric	→	`met:hasName`
			value	→	`met:hasValue`

5.4.4 Discussion

In our case study, we showed how ONTOACCESS has been successfully deployed to *make a bridge to the new town*. It provides a gradual migration path from a legacy system such as EVOLIZER, to a new platform, in our example SOFAS. We demonstrated how ONTOACCESS bridges the conceptual gap between the relational data model of EVOLIZER and the RDF-based SOFAS. We further motivated that existing, read-only RDB-to-RDF mapping approaches are unsuitable for this application scenario as they limit RDF-based data access to read-only queries. During this case study, we faced several challenges *w.r.t.* to the mapping in ONTOACCESS. In the following, we report on two major ones and the solutions we developed to overcome them.

Table 5.4: Mapping Overview Part II

link table	→ property	| inverse property
Release_Revision	→ `ver:comprises`	| `ver:appearsIn`
Branch_Revision	→ `ver:comprises`	| `ver:isOn`
Transaction_Revision	→ `ver:comprises`	| `ver:commitedIn`
File_Revision	→ `ver:hasVersion`	| `ver:belongsTo`
Person_Revision	→ -	| `ver:committedBy`
Class_Revision	→ `ver:hasSource`	| -
Method_Class	→ `java:isDeclaredMethodOf`	| `java:declaresMethod`
Attribute_Class	→ `java:isDeclaredFieldOf`	| `java:declaresField`
Measurement_Entity	→ `met:isMetricOf`	| `met:hasMetric`
Inheritance	→ `java:hasSubClass`	| `java:hasSuperClass`
Invocation	→ `java:invokesMethod`	| `java:isInvokedByMethod`
Access	→ `java:accessField`	| `java:isAccessedByMethod`

The first challenge is related to the representation of concept inheritance in relational database systems. Inheritance is a central concept in the object-oriented methodology and is therefore commonly used in object-oriented systems, including EVOLIZER. Relational, unlike object-relational or object-oriented databases, do not directly support inheritance. However, there exist three principal strategies to implement inheritance in relational database schemata (*cf.* [Garcia-Molina et al., 2008]): *table-per-hierarchy* represents all classes of the inheritance hierarchy in a single table. This table contains columns for the attributes of all classes and a special column, called discriminator, that stores the type (*i.e.*, class) for each instance. *Table-per-concrete-class* represents each class in its own table. Each of those tables contains columns for the attributes of the class and all super-classes up to the root of the inheritance hierarchy. As a result, attributes of a common super-class are duplicated in all of its sub-classes. The third strategy, called *table-per-subclass*, also represents each class in its own table. In contrast to the *table-per-concrete-class* strategy, the attributes of the super-class(es) are not duplicated as columns in the sub-classes. Instead, a shared primary key is used to connect the tables representing classes in the inheritance hierarchy.

EVOLIZER uses different strategies for different inheritance hierarchies, for example the *table-per-hierarchy* strategy to implement inheritance for the *Entity* concept and its subconcepts. For the sake of this case study, we had to add explicit support for mapping inheritance hierarchies to ONTOACCESS. The *table-per-concrete-class* strategy was mappable out-of-the-box since it defines one table per class and the tables are independent from each other. Mapping the other two strategies required support for features such as discriminator columns and relating tables in a parent-child relationship. We addressed this limitation by adding explicit mapping constructs to the ONTOACCESS mapping language. First, discriminator columns were added to provide support for the *table-per-hierarchy* strategy. Since support for mapping a subset of the columns in a table already exists, it is possible to provide multiple mappings for tables that represent all classes within an inheritance hierarchy (one mapping for each class). Each mapping only contains the respective subset of the columns and a description of the discriminator column with its name and value. Listing 5.2a) depicts a concrete mapping example that is using a discriminator column. We also added a mapping construct for relating two tables to each other in a parent-child relationship to provide support for the *table-per-subclass* strategy. The mapping of a table can reference another table as its parent table. This enables ONTOACCESS to detect that a concept from the application domain is split among multiple tables in the database schema. As a result, the involved tables can automatically be joined (on the primary key). Listing 5.2b) depicts a concrete mapping example that is using a parent table reference. These two extensions to R3M enable support for mapping the relational representations of concept inheritance with all three strategies.

The second challenge is related to defining the RDB-to-RDF mappings. Mappings in ONTOACCESS are encoded in RDF which makes them well-suited for automatic processing by machines but hinders the accessibility for human users. Manually defining such mappings is a time-consuming and error-prone task, consisting of mostly repetitive steps. Therefore, tool support for defining mapping is indispensable in more complex application scenarios where the number of database tables and columns is of significance. We built a

```
a)  ex:method  a  r3m:TableMap;
      r3m:hasTableName      "Entity";
      r3m:mapsToClass       java:Method;
      r3m:hasDiscriminator  ex:method_type;
      r3m:uriPattern        "http://.../method_%%id%%";
      r3m:hasAttribute      ex:method_type, ....
    ex:method_type  a  r3m:AttributeMap;
      r3m:hasAttributeName  "ctype";
      r3m:hasValue          "Method".

b)  ex:Method  a  r3m:TableMap;
      r3m:hasTableName    "Method";
      r3m:mapsToClass     java:Method;
      r3m:hasParentTable  ex:entity;
      r3m:uriPattern      "http://.../method_%%id%%";
      r3m:hasAttribute    ex:method_returnType.
    ex:entity  a  r3m:TableMap;
      r3m:hasTableName    "Entity";
      r3m:hasAttribute    ex:entity_uniqueName, ....
```

Listing 5.2: Extended R3M Mapping Examples

tool [Brügger, 2009] to ease the definition of ONTOACCESS mappings. It semi-automatically generates a mapping from an RDB schema in two steps. First, it automatically generates a basic mapping, based on information extracted from the schema catalog of the database system. Terms of the target ontology are also generated in this step, based on table and column names in the database schema. Next, the tool displays a graphical editor for refining the mapping. This step is mainly concerned with replacing the generated terms with actual terms from the target ontology. The tool further provides validation of existing mappings to catch errors from manual editing. The tool is implemented as a plug-in for the ontology editor Protégé[4] to enable quick access to the definition of the target ontology.

[4] http://protege.stanford.edu/

5.5 Related Work

D2R Server [Bizer and Cyganiak, 2006] is an approach for publishing existing relational databases on the Semantic Web. Based on mappings expressed in the D2RQ [Bizer and Seaborne, 2004] mapping language, it enables browsing the relational data as RDF via dereferenceable URIs (*i.e.*, as Linked Data). Further, support for the SPARQL query language is provided. D2R is limited to read-only data access, updating RDF data is not supported.

The Virtuoso Universal Server features RDF Views [Erling and Mikhailov, 2007] to expose relational data on the Semantic Web. A declarative Meta Schema Language is used for defining the mapping of SQL data to RDF vocabularies. This enables the use of SPARQL as an alternative query language for the relational data. Likewise, Virtuoso implements a Linked Data interface to these views. RDF Views are limited to read-only queries.

R_2O [Barrasa et al., 2004] is an extensible and fully declarative language to describe mappings between relational database schemata and ontologies. R_2O is aimed at situations where the similarity between the ontology and the database model is low. It has been conceived to be expressive enough to cope with complex mapping cases where one model is richer, more generic/specific, or better structured than the other. This high expressiveness renders R_2O mappings read-only.

The W3C has recognized the importance of mapping relational data to the Semantic Web by starting the RDB2RDF Incubator Group[5] (XG) to investigate the need for standardization. The XG recommended [Malhotra, 2009] that the W3C initiates a working group (WG) to define a vendor-independent RDB-to-RDF mapping language. The RDB2RDF WG[6] started its work on R2RML [Das et al., 2010] in late 2009. According to their charter [Halpin and Herman, 2009], the requirements for updating relational data are out of scope and are therefore not addressed by the WG. It was shown in [Garrote and Garcia, 2011] that adding write support to the R2RML approach is impractical.

[5] `http://www.w3.org/2005/Incubator/rdb2rdf/`
[6] `http://www.w3.org/2001/sw/rdb2rdf/`

5.6 Conclusions

In theory the Semantic Web provides a common framework that greatly facilitates data sharing and reuse across application, enterprise, and community boundaries. In practice its wide adoption is still hampered by the fact that many organizations have locked away their data in relational databases. Business-critical legacy applications rely on these databases to sustain daily operations and newly developed systems often need to run in tandem with their predecessors until the latter can be gradually phased out. Both, the legacy systems, as well as their successors, usually need to operate cooperatively on existing data. This includes reads and updates. A complete paradigm shift in data representation is therefore often extremely difficult and costly to achieve.

In this paper, we presented OntoAccess, an RDB-to-RDF mediation platform that enables RDF-based read and write access to relational databases. It greatly facilitates the transition from legacy systems to Semantic Web-enabled applications in practice by providing a semantic layer on top of existing relational databases. Semantic Web query and update requests are translated on-the-fly to SQL for execution in the database system. OntoAccess therefore eliminates the need for mirroring and synchronizing relational data with its RDF representation and, in addition, allows one to further exploit the advantages of the well-established database technology, such as query performance, scalability, transaction support, and security.

In our case study, we have described how we successfully deployed OntoAccess to provide a gradual migration path between two of our own large-scale software systems, namely the legacy application Evolizer and its successor, the Sofas platform. We identified challenges when it comes to mapping inheritance hierarchies with OntoAccess and we have extended the latter accordingly to support different inheritance mapping strategies. Further, we established tooling to semi-automate the process of extracting mappings from a relational database schema to an ontology.

In summary, judging from the experiences made in our case study, we are confident that OntoAccess is a valuable tool that will foster the acceptance of Semantic Web technology in practice.

A

Publications

This appendix presents the list of publications this dissertation is based on.

A.1 Journal Article

[Hert et al., 2012] Matthias Hert, Gerald Reif, and Harald C. Gall. A Platform for RDF-based Read an Write Access to Relational Databases. Submitted for Publication to *Journal of Web Semantics: Science, Services and Agents on the World Wide Web*, 2012. *(Under Review).*

A.2 Conference Papers

[Hert et al., 2011b] Matthias Hert, Sergio Marsella, Gerald Reif, and Harald C. Gall. UpLink – A Linked Data Editor for RDB-to-RDF Data. In *Proceedings of the 7th International Conference on Semantic Systems*, September 2011.

[Hert et al., 2011c] Matthias Hert, Gerald Reif, and Harald C. Gall. A Comparison of RDB-to-RDF Mapping Languages. In *Proceedings of the 7th International Conference on Semantic Systems*, 2011.

[Hert et al., 2011a] Matthias Hert, Giacomo Ghezzi, Michael Würsch, and Harald C. Gall. How to "Make a Bridge to the New Town" using OntoAccess. In *Proceedings of the 10th International Semantic Web Conference*, October 2011. *(Nominated for Best In Use Paper Award).*

A.3 PhD Symposium

[Hert, 2009] Matthias Hert. Relational Databases as Semantic Web Endpoints. In *Proceedings of the 6th European Semantic Web Conference*, May/ June 2009.

A.4 Workshop Papers

[Hert et al., 2010b] Matthias Hert, Gerald Reif, and Harald C. Gall. Updating Relational Data via SPARQL/Update. In *EDBT Workshop Proceedings*, March 2010.

[Hert et al., 2010a] Matthias Hert, Gerald Reif, and Harald C. Gall. 'Semantic Web 2.0' – Write-enabling the Web of Data. In *Proceedings of the 6th Workshop on Semantic Web Applications and Perspectives*, September 2010.

A.5 Poster

[Hert et al., 2009] Matthias Hert, Gerald Reif, and Harald C. Gall. SPARQL/Update for Relational Databases. In *Proceedings of the 6th European Semantic Web Conference*, May/June 2009.

Bibliography

[Angles and Gutierrez, 2008] Angles, R. and Gutierrez, C. (2008). The Expressive Power of SPARQL. In *Proceedings of the 7th International Semantic Web Conference*.

[Auer et al., 2009] Auer, S., Dietzold, S., Lehmann, J., Hellmann, S., and Aumueller, D. (2009). Triplify – Light-Weight Linked Data Publication from Relational Databases. In *Proceedings of the 18th International World Wide Web Conference*.

[Bancilhon and Spyratos, 1981] Bancilhon, F. and Spyratos, N. (1981). Update Semantics of Relational Views. *ACM Transactions on Database Systems*.

[Barrasa et al., 2003] Barrasa, J., Corcho, O., and Gómez-Pérez, A. (2003). Fund Finder: A Case Study of Database-to-Ontology Mapping. In *Proceedings of the Semantic Integration Workshop*.

[Barrasa et al., 2004] Barrasa, J., Corcho, O., and Gómez-Pérez, A. (2004). R2O, an Extensible and Semantically Based Database-to-Ontology Mapping Language. In *Proceedings of the 2nd Workshop on Semantic Web and Databases*.

[Beckett and Berners-Lee, 2011] Beckett, D. and Berners-Lee, T. (2011). Turtle – Terse RDF Triple Language. W3C Team Submission `http://www.w3.org/TeamSubmission/turtle/`.

[Beckett and Grant, 2003] Beckett, D. and Grant, J. (2003). SWAD-Europe Deliverable 10.2: Mapping Semantic Web Data with RDBM-

Ses. http://www.w3.org/2001/sw/Europe/reports/scalable_rdbms_mapping_report/. Last visited July 2011.

[Berners-Lee, 2009a] Berners-Lee, T. (2009a). Linked Data. http://www.w3.org/DesignIssues/LinkedData.html. Last visited June 2011.

[Berners-Lee, 2009b] Berners-Lee, T. (2009b). Relational Databases on the Semantic Web. http://www.w3.org/DesignIssues/RDB-RDF.html. Last visited July 2011.

[Berners-Lee et al., 2009] Berners-Lee, T., Cyganiak, R., Hausenblas, M., Presbrey, J., Seneviratne, O., and Ureche, O.-E. (2009). Realising a Read-Write Web of Data. http://web.mit.edu/presbrey/Public/rw-wod.pdf. Last visited July 2011.

[Berners-Lee et al., 2001] Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The Semantic Web. *Scientific American*.

[Biron and Malhotra, 2004] Biron, P. V. and Malhotra, A. (2004). XML Schema Part 2: Datatypes Second Edition. W3C Recommendation. http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/.

[Bizer, 2003] Bizer, C. (2003). D2R MAP – A Database to RDF Mapping Language. In *Proceedings of the 12th International World Wide Web Conference*.

[Bizer and Cyganiak, 2006] Bizer, C. and Cyganiak, R. (2006). D2R Server – Publishing Releational Databases on the Semantic Web. In *Proceedings of the 5th International Semantic Web Conference*.

[Bizer et al., 2009] Bizer, C., Cyganiak, R., Garbers, J., Maresch, O., and Becker, C. (2009). The D2RQ Platform v0.7 – Treating Non-RDF Relational Databases as Virtual RDF Graphs - User Manual and Language Specification. http://www4.wiwiss.fu-berlin.de/bizer/d2rq/spec/20090810/.

[Bizer et al., 2011] Bizer, C., Heath, T., Berners-Lee, T., and Hausenblas, M. (2011). Linked Data on the Web – Topics of Interest. http://events.linkeddata.org/ldow2011/. Last visited July 2011.

[Bizer and Schultz, 2008a] Bizer, C. and Schultz, A. (2008a). Benchmarking the Performance of Storage Systems that Expose SPARQL Endpoints. In *Proceedings of the 4th International Workshop on Scalable Semantic Web Knowledge Base Systems*.

[Bizer and Schultz, 2008b] Bizer, C. and Schultz, A. (2008b). Berlin SPARQL Benchmark (BSBM) Specification – V2.0. `http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/spec/`.

[Bizer and Schultz, 2009] Bizer, C. and Schultz, A. (2009). The Berlin SPARQL Benchmark. *International Journal on Semantic Web and InformationSystems*.

[Bizer and Seaborne, 2004] Bizer, C. and Seaborne, A. (2004). D2RQ – Treating Non-RDF Databases as Virtual RDF Graphs. In *Proceedings of the 3rd International Semantic Web Conference*.

[Bohannon et al., 2006] Bohannon, A., Pierce, B. C., and Vaughan, J. A. (2006). Relational Lenses: A Language for Updatable Views. In *Proceedings of the 25th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*.

[Brügger, 2009] Brügger, N. (2009). RDB-RDF Mapping Generation from Relational Database Schemata. Master's thesis, University of Zurich.

[Calvanese et al., 2007a] Calvanese, D., Giacomo, G. D., Lembo, D., Lenzerini, M., Poggi, A., and Rosati, R. (2007a). MASTRO-I: Efficient Integration of Relational Data through DL Ontologies. In *Proceedings of the 20th International Workshop on Description Logics*.

[Calvanese et al., 2007b] Calvanese, D., Giacomo, G. D., Lembo, D., Lenzerini, M., and Rosati, R. (2007b). Tractable Reasoning and Efficient Query Answering in Description Logic: The DL-Lite Family. *Journal of Automated Reasoning*.

[Carroll et al., 2004] Carroll, J. J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., and Wilkinson, K. (2004). Jena: Implementing the Semantic

Web Recommendations. In *Proceedings of the 13th International World Wide Web Conference*.

[Chang et al., 2004] Chang, K. C.-C., He, B., Li, C., Patel, M., and Zhang, Z. (2004). Structured Databases on the Web: Observations and Implications. *SIGMOD Record*.

[Das et al., 2010] Das, S., Sundara, S., and Cyganiak, R. (2010). R2RML: RDB to RDF Mapping Language. W3C Working Draft. `http://www.w3.org/TR/2010/WD-r2rml-20101028/`.

[Dayal and Bernstein, 1982] Dayal, U. and Bernstein, P. A. (1982). On the Correct Translation of Update Operations on Relational Views. In *ACM Transactions on Database Systems*.

[de Laborda and Conrad, 2005] de Laborda, C. P. and Conrad, S. (2005). Relational.OWL – A Data and Schema Representation Format Based on OWL. In *Proceedings of the 2nd Asia-Pacific Conference on Conceptual Modelling*.

[de Laborda et al., 2006] de Laborda, C. P., Zloch, M., and Conrad, S. (2006). RDQuery – Querying Relational Databases on-the-fly with RDF-QL. In *Proceedings of the 15th International Conference on Knowledge Engineering and Knowledge Management*.

[Demeyer et al., 2002] Demeyer, S., Ducasse, S., and Nierstrasz, O. (2002). *Object Oriented Reengineering Patterns*. Morgan Kaufmann Publishers Inc.

[Erling and Mikhailov, 2007] Erling, O. and Mikhailov, I. (2007). RDF Support in the Virtuoso DBMS. In *Proceedings of the SABRE Conference on Social Semantic Web*.

[Fielding, 2000] Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine.

[Fischer et al., 2003] Fischer, M., Pinzger, M., and Gall, H. (2003). Populating a Release History Database from Version Control and Bug Tracking Systems. In *Proceedings of the 19th International Conference on Software Maintenance*.

[Fürber, 2009] Fürber, C. (2009). Ontology-Based Data Quality Management: Methodology, Cost, and Benefits. In *Proceedings of the 6th European Semantic Web Conference*.

[Fürber and Hepp, 2010] Fürber, C. and Hepp, M. (2010). Using SPARQL and SPIN for Data Quality Management on the Semantic Web. In *Proceedings of the 13th International Conference on Business Information Systems*.

[Gall et al., 2009] Gall, H. C., Fluri, B., and Pinzger, M. (2009). Change Analysis with Evolizer and ChangeDistiller. *IEEE Software*.

[Garcia-Molina et al., 2008] Garcia-Molina, H., Ullman, J. D., and Widom, J. (2008). *Database Systems: The Complete Book*. Prentice Hall Press.

[Garrote and Garcia, 2011] Garrote, A. and Garcia, M. N. M. (2011). RESTful Writable APIs for the Web of Linked Data Using Relational Storage Solutions. In *Proceedings of the WWW2011 Workshop on Linked Data on the Web*.

[Ghezzi and Gall, 2008] Ghezzi, G. and Gall, H. C. (2008). Towards Software Analysis as a Service. In *Proceedings of the 4th International ERCIM Workshop on Software Evolution and Evolvability*.

[Ghezzi and Gall, 2011] Ghezzi, G. and Gall, H. C. (2011). SOFAS: A Lightweight Architecture for Software Analysis as a Service. In *Proceedings of the 9th Working IEEE/IFIP Conference on Software Architecture*.

[Halpin and Herman, 2009] Halpin, H. and Herman, I. (2009). RDB2RDF Working Group Charter. `http://www.w3.org/2009/08/rdb2rdf-charter`. Last visited July 2011.

[Hayes, 2004] Hayes, P. (2004). RDF Semantics. W3C Recommendation. `http://www.w3.org/TR/2004/REC-rdf-mt-20040210/`.

[Hert, 2009] Hert, M. (2009). Relational Databases as Semantic Web Endpoints. In *Proceedings of the 6th European Semantic Web Conference*.

[Hert et al., 2011a] Hert, M., Ghezzi, G., Würsch, M., and Gall, H. C. (2011a). How to "Make a Bridge to the New Town" using OntoAccess. In *Proceedings of the 10th International Semantic Web Conference*. (Nominated for Best In Use Paper Award).

[Hert et al., 2011b] Hert, M., Marsella, S., Reif, G., and Gall, H. C. (2011b). UpLink – A Linked Data Editor for RDB-to-RDF Data. In *Proceedings of the 7th International Conference on Semantic Systems*.

[Hert et al., 2009] Hert, M., Reif, G., and Gall, H. C. (2009). SPARQL/Update for Relational Databases. In *Proceedings of the 6th European Semantic Web Conference*.

[Hert et al., 2010a] Hert, M., Reif, G., and Gall, H. C. (2010a). 'Semantic Web 2.0' – Write-enabling the Web of Data. In *Proceedings of the 6th Workshop on Semantic Web Applications and Perspectives*.

[Hert et al., 2010b] Hert, M., Reif, G., and Gall, H. C. (2010b). Updating Relational Data via SPARQL/Update. In *EDBT Workshop Proceedings*.

[Hert et al., 2011c] Hert, M., Reif, G., and Gall, H. C. (2011c). A Comparison of RDB-to-RDF Mapping Languages. In *Proceedings of the 7th International Conference on Semantic Systems*.

[Hert et al., 2012] Hert, M., Reif, G., and Gall, H. C. (2012). OntoAccess – An Extensible Platform for RDF-based Read and Write Access to Relational Databases. *Journal of Web Semantics: Science, Services and Agents on the World Wide Web*. (Under Review).

[Hu and Qu, 2007] Hu, W. and Qu, Y. (2007). Discovering Simple Mappings Between Relational Database Schemas and Ontologies. In *Proceedings of the 6th International and 2nd Asian Semantic Web Conference*.

[ISO/IEC, 2006] ISO/IEC (2006). ISO/IEC 9075 Part 14: XML-Related Specifications (SQL/XML).

[Keller, 1985] Keller, A. M. (1985). Algorithms for Translating View Updates to Database Updates Involving Selections, Projections, and Joins. In *Proceedings of the 4th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*.

[Langegger et al., 2008] Langegger, A., Wöss, W., and Blöchl, M. (2008). A Semantic Web Middleware for Virtual Data Integration on the Web. In *Proceedings of the 5th European Semantic Web Conference*.

[Langerak, 1990] Langerak, R. (1990). View Updates in Relational Databases with an Independent Scheme. In *ACM Transactions on Database Systems*.

[Ma et al., 2009] Ma, L., Sun, X., Cao, F., Wang, C., and Wang, X. (2009). Semantic Enhancement for Enterprise Data Management. In *Proceedings of the 8th International Semantic Web Conference*.

[Malhotra, 2009] Malhotra, A. (2009). W3C RDB2RDF Incubator Group Report. `http://www.w3.org/2005/Incubator/rdb2rdf/XGR-rdb2rdf-20090126/`.

[Manola and Miller, 2004] Manola, F. and Miller, E. (2004). RDF Primer. W3C Recommendation. `http://www.w3.org/TR/2004/REC-rdf-primer-20040210/`.

[Motik et al., 2009] Motik, B., Grau, B. C., Horrocks, I., Wu, Z., Fokoue, A., and Lutz, C. (2009). OWL 2 Web Ontology Language Profiles. W3C Recommendation. `http://www.w3.org/TR/owl-profiles/`.

[Ogbuji, 2011] Ogbuji, C. (2011). SPARQL 1.1 Graph Store HTTP Protocol. W3C Working Draft. `http://www.w3.org/TR/2011/WD-sparql11-http-rdf-update-20110512/`.

[OpenLink Software, 2008] OpenLink Software (2008). Mapping Relational Data to RDF with Virtuoso's RDF Views. `http://virtuoso.openlinksw.`

com/whitepapers/relational%20rdf%20views%20mapping.html. Last visited July 2011.

[Patel et al., 2009] Patel, C., Khan, S., and Gomadam, K. (2009). TrialX: Using Semantic Technologies to Match Patients to Relevant Clinical Trials Based on Their Personal Health Records. In *Proceedings of the 8th International Semantic Web Conference*.

[Prud'hommeaux and Hausenblas, 2010] Prud'hommeaux, E. and Hausenblas, M. (2010). Use Cases and Requirements for Mapping Relational Databases to RDF. W3C Working Draft. http://www.w3.org/TR/2010/WD-rdb2rdf-ucr-20100608/.

[Prud'hommeaux and Seaborne, 2008] Prud'hommeaux, E. and Seaborne, A. (2008). SPARQL Query Language for RDF. W3C Recommendation. http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/.

[Sahoo et al., 2009] Sahoo, S. S., Halb, W., Hellmann, S., Idehen, K., Jr, T. T., Auer, S., Sequeda, J., and Ezzat, A. (2009). A Survey of Current Approaches for Mapping of Relational Databases to RDF. http://www.w3.org/2005/Incubator/rdb2rdf/RDB2RDF_SurveyReport.pdf. Last visited July 2011.

[Schenk et al., 2010] Schenk, S., Gearon, P., and Passant, A. (2010). SPARQL 1.1 Update. W3C Working Draft. http://www.w3.org/TR/2010/WD-sparql11-update-20101014/.

[Seaborne et al., 2008] Seaborne, A., Manjunath, G., Bizer, C., Breslin, J., Das, S., Davis, I., Harris, S., Idehen, K., Corby, O., Kjernsmo, K., and Nowack, B. (2008). SPARQL Update – A Language for Updating RDF Graphs. W3C Member Submission. http://www.w3.org/Submission/2008/SUBM-SPARQL-Update-20080715/.

[SquirrelRDF, 2006] SquirrelRDF (2006). SquirrelRDF. http://jena.sourceforge.net/SquirrelRDF/. Last visited July 2011.

[World Wide Web Consortium, 2011] World Wide Web Consortium (2011). W3C Semantic Web Activity. http://www.w3.org/2001/sw/.

Curriculum Vitae

Personal Information

Name	Matthias Hert
Nationality	Swiss
Date of Birth	February 27, 1983
Place of Birth	Messen, Switzerland

Education

2008 – 2012	PhD in Informatics *(magna cum laude)* Department of Informatics University of Zurich, Switzerland
2003 – 2008	MSc in Informatics *(summa cum laude)* Department of Informatics University of Zurich, Switzerland
2002 – 2003	Student of Computer Science Swiss Federal Institute of Technology Zurich, Switzerland
1997 – 2002	Matura Typus C Kantonschule Solothurn, Switzerland

OntoAccess

RDF-based Read and Write Access to Relational Databases

Relational Databases (RDBs) are used in most current enterprise environments to store and manage data. While RDBs are well suited to handle large amounts of data, they were not designed to preserve the data semantics. The meaning of the data is implicit at the application level but not explicitly encoded in the relational model. The Semantic Web provides a common framework that allows data to be shared and reused across application, enterprise, and community boundaries. Although developed for the Web, these Semantic Web technologies have proven to be useful in other domains as well, especially if data from different sources has to be exchanged or integrated. In existing systems, however, it is not always possible or desirable to convert all relational data to RDF as other business-critical applications rely on the *relational* representation of the data. Adapting or replacing these applications would require a prohibitive migration effort. Therefore, a mediation approach is needed that bridges the conceptual gap between the relational model and RDF, resulting in a cooperative use of the data in RDF-based as well as relational applications.

In the past, various RDB-to-RDF mediation approaches were explored, resulting in the definition of multiple RDB-to-RDF mappings and algorithms to translate Semantic Web queries to the RDB. However, all of these approaches are limited to read-only data access and have a strong focus on SPARQL for querying and Linked Data for browsing the data as RDF. Use cases where write access to the RDB or support for other data access approaches is needed have so far been neglected by the state-of-the-art RDB-to-RDF mediation approaches.

In this dissertation we present OntoAccess, a RDB-to-RDF mediation approach that enables RDF-based read and write access to a RDB. The approach consists of three parts: (1) the RDB-to-RDF mapping called R3M that provides the basis for RDF-based read and write access to the RDB; (2) algorithms to translate RDF-based read and write requests to the RDB; and (3) an architecture for an extensible RDB-to-RDF mediation that enables support for multiple data access approaches.

To validate our OntoAccess approach for RDB-to-RDF mediation we provide the following: (1) a formal definition of our RDB-to-RDF mapping R3M and proofs of its bidirectional properties; (2) a performance evaluation of our algorithms for translating RDF-based requests to the RDB; (3) a proof of concept implementation of our architecture for an extensible RDB-to-RDF mediation platform; and (4) a case study in the domain of software analysis where we apply OntoAccess to make a data bridge between a RDB-based legacy system and its RDF-based long-term replacement.

In summary, we therefore state: *The OntoAccess approach, consisting of a mapping, an architecture, and algorithms, bridges the conceptual gap between the relational data model and RDF and therefore enables RDF-based read and write access to a RDB.*